

AFRL-IF-RS-TR-2003-29
Final Technical Report
February 2003



TOOLS FOR ASSEMBLING AND MANAGING SCALABLE KNOWLEDGE BASES

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F109

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.


AFRL-IF-RS-TR-2003-29 has been reviewed and is approved for publication.

APPROVED:



RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR:



JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE FEBRUARY 2003	3. REPORT TYPE AND DATES COVERED Apr 97 – May 02	
4. TITLE AND SUBTITLE TOOLS FOR ASSEMBLING AND MANAGING SCALABLE KNOWLEDGE BASES			5. FUNDING NUMBERS C - F30602-97-1-0194 PE - 62301E PR - IIST TA - 00 WU - 09	
6. AUTHOR(S) Hans Chalupsky				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California Information Science Institute 4676 Admiralty Way Marina Del Rey California 90292			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFTB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-29	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577/ Raymond.Liuzzi@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) The DARPA High Performance Knowledge Base (HPKB) program was aimed to produce technology to rapidly construct large, reusable, and maintainable ontologies and knowledge bases (KBs). To achieve this goal, large-scale KBs cannot always be built from scratch, but instead need to be assembled as much as possible from existing resources. Reuse, however, does not come for free: reusable material has to be identified, translated, adapted, debugged, merged with other material and maintained, all of which can be very difficult and expensive processes. Therefore, for reuse to be effective, it has to be supported by a set of adequate knowledge base construction, editing and maintenance tools. This report describes an HPKB effort that built a variety of tools and infrastructure aimed at supporting the ontology and knowledge base construction process. All these tools are centered around the PowerLoom Knowledge Representation and Reasoning (KR&R) system (http://www.isi.edu/isd/LOOM/PowerLoom), which is a highly expressive, logic-based KR&R system with multiple built-in deductive reasoning capabilities including a query processor, a description classifier, and a context mechanism. The developed tools cover various areas of the knowledge base and ontology construction process and are outlined in the report.				
14. SUBJECT TERMS Computers, Knowledge Base, Artificial Intelligence, Software, Reasoning				15. NUMBER OF PAGES 85
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNCLASSIFIED	

Table of Contents

1 TOOLS FOR ASSEMBLING AND MANAGING SCALABLE KNOWLEDGE BASES	1
1.1 Knowledge Translation	1
1.2 Browsing and Editing.....	2
1.3 Learning and Partial Inference	3
1.4 Large-scale Persistent Storage and Scalable Inference.....	4
1.5 Other Tools	6
2 THE ONTOMORPH TRANSLATOR FOR SYMBOLIC KNOWLEDGE	8
2.1 Introduction	8
2.2 The Translation Problem.....	9
2.3 Dimensions of Mismatch	10
2.4 OntoMorph	12
2.4.1 Syntactic Rewriting	12
2.4.2 Semantic Rewriting	18
2.5 OntoMorph Applications.....	20
2.5.1 Course of Action Critiquer	20
2.5.2 Rosetta Agent Translation Service.....	24
2.6 Related Work.....	27
2.7 Future Work: Ontology Merging.....	28
3 DETERMINING DECISIVE POINTS THROUGH CASE-BASED REASONING	30
3.1 Overview.....	30
3.2 The Decisive Point Problem.....	31
3.2.1 Problem Description	31
3.2.2 A Case-Based Reasoning Solution	32
3.2.3 Technical Hurdles.....	34
3.3 Knowledge-Rich Case-Based Reasoning	36
3.3.1 Generating Match Criteria	38
3.3.2 Building Semantic Signatures.....	41
3.3.3 Incorporating Rule Knowledge.....	44
3.4 Evaluation	45

3.5 Conclusions	48
4 JAVA-BASED GRAPHICAL KNOWLEDGE EDITOR	50
4.1 Overview.....	50
4.2 Architecture	50
4.2.1 Client/Server Architecture	50
4.2.2 GUI Design Goals.....	51
4.3 GUI Overview	55
4.4 GUI Features.....	58
4.4.1 Connect to Server	58
4.4.2 Edit Preferences	58
4.4.3 KB Load/Save.....	59
4.4.4 Browsing.....	59
4.4.5 Editing/Viewing.....	64
4.4.6 Choosers	67
4.4.7 Extension Editor	68
4.4.8 Query/Ask.....	70
4.4.9 Search	71
4.4.10 Console	72
4.4.11 Cut/Copy/Paste/Delete.....	73
4.5 Future Work	74
4.5.1 Large KBs.....	74
4.5.2 Drag/Drop	74
4.5.3 Scrapbook	74
4.5.4 Instance cloning	75
4.5.5 Security	75
4.5.6 Multiple users	75
5 REFERENCES	76

List of Figures

Figure 1: OntoSaurus browser for PowerLoom knowledge bases.....	3
Figure 2: The knowledge translation problem.	9
Figure 3: Syntax tree representation of $f(\mathcal{G}([x], y))$	13
Figure 4: Case-based Reasoning High Level Algorithm. In the decisive point problem, x refers to a decisive point description and $f(x)$ refers to the goodness of the decisive point.	33
Figure 5: A partial decisive point description.....	35
Figure 6: Decisive point algorithm	37
Figure 7: A generalized knowledge structure. Variables are substituted for all non-leaf instances.....	39
Figure 8: An example of structure folding in GSA. The figure illustrates folding three different knowledge structures into one unified structure. The first structure is simply copied to the empty unified structure. In the second example, variable $?V4$ is unified with variable $?V1$, since they both share the relation $r1$ from $?X$	40
Figure 9: Transformation of a structural case into a floating point signature.....	42
Figure 10: Learning curves for different variations of the case-based reasoner.....	47
Figure 11: The PowerLoom GUI.....	55
Figure 12: Knowledge Browser.....	60
Figure 13: Instance Editor.....	65
Figure 14: Proposition Editor.....	67
Figure 15: Extension Editor.....	68
Figure 16: Query Dialog.....	69
Figure 17: Search Dialog	71
Figure 18: PowerLoom Console	73

List of Tables

Table 1: Results from the HPKB evaluation. Scores range from 0 to 100.	46
---	----

1 Tools for Assembling and Managing Scalable Knowledge Bases

The HPKB program was aimed to produce technology to rapidly construct large, reusable, and maintainable ontologies and knowledge bases (KBs). To achieve this goal, large-scale KBs cannot always be built from scratch, but instead need to be assembled as much as possible from existing resources. Reuse, however, does not come for free: reusable material has to be identified, translated, adapted, debugged, merged with other material and maintained, all of which can be very difficult and expensive processes. Therefore, for reuse to be effective, it has to be supported by a set of adequate knowledge base construction, editing and maintenance tools.

As part of our participation in HPKB we built a variety of tools and infrastructure aimed at supporting the ontology and knowledge base construction process. All these tools are centered around the PowerLoom knowledge representation and reasoning (KR&R) system (<http://www.isi.edu/isd/LOOM/PowerLoom>), which is a highly expressive, logic-based KR&R system with multiple built-in deductive reasoning capabilities including a query processor, a description classifier, and a context mechanism. The developed tools cover various areas of the knowledge base and ontology construction process and are outlined in more detail below.

1.1 Knowledge Translation

As mentioned above, reuse is a very important prerequisite for being able to build large, high performance knowledge bases as quickly as possible. However, relevant ontologies and knowledge bases that could be reused are often formulated in a different knowledge representation language than the one that is required, use different or incompatible modeling conventions, or need to be semantically altered or “morphed” to fit with the newly developed knowledge base. Therefore, one needs a powerful translation tool that allows one to easily translate and adapt reusable knowledge into the required format before it can be integrated with other parts of the new knowledge base.

To support this part of the ontology and knowledge base construction process, we built the OntoMorph translation tool for symbolic knowledge. OntoMorph provides a

powerful rule language to represent complex syntactic transformations, and it is fully integrated with the PowerLoom KR&R system to allow transformations based on any mixture of syntactic and semantic criteria. Within HPKB, OntoMorph was first successfully applied as the input translator for the course of action critiquing system developed by the Expect group at USC ISI. Since then it has been used in other applications such as a translation service for agent communication as well as file-based and dynamic CycL to PowerLoom translators. Details of OntoMorph are described in Section 2, which also motivates how OntoMorph can be used to support knowledge base merging tasks.

1.2 Browsing and Editing

Another important part of the knowledge base construction process are powerful browsing and editing tools. These tools enable the knowledge engineer to easily navigate through potentially very large ontologies and knowledge bases, visualize and understand their structure and then make any necessary modifications.

To support the browsing process, we adapted the Web-based OntoSaurus knowledge base browser [Swartout et. al, 1996] to be able to display and navigate through PowerLoom knowledge bases. OntoSaurus was initially developed for the Loom KR&R system (PowerLoom's predecessor) which is a description logic system and uses a representation language quite different from the one used by PowerLoom. Therefore, we had to reimplement OntoSaurus to work with PowerLoom knowledge bases. The new version of OntoSaurus was written in the STELLA programming language [Chalupsky & MacGregor, 1999] which allows us to deliver it in Lisp, C++ and Java versions. A screen shot of OntoSaurus displaying parts of a PowerLoom knowledge base is shown in Figure 1.

A browsing tool is of course not enough, one also needs to be able to edit and modify the knowledge base. To that end, we built a completely new Java-based knowledge base editing tool for PowerLoom. While initially we planned to extend OntoSaurus with the required editing functions, we finally decided to implement the editor directly in Java, since that gives much higher flexibility, better support for complex editing functions such as context dependent cut and paste, completion, etc. Using new technology such as Java

WebStart, we can continue to give users the ability to launch the interface by simply using their Web browser while still reaping the flexibility benefits of writing the editor directly in Java. More details of the Java-based knowledge editor are described in Section 4.

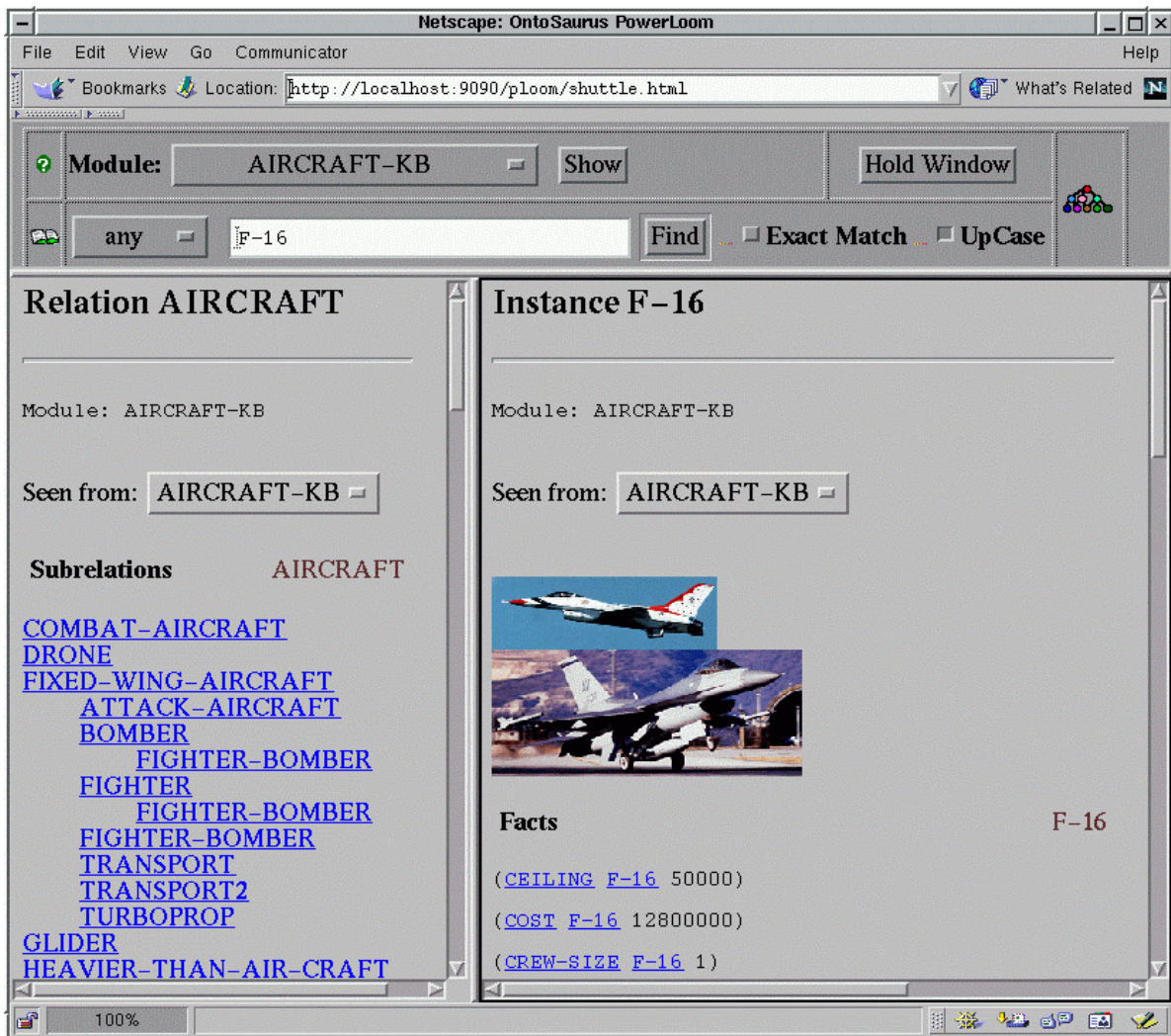


Figure 1: OntoSaurus browser for PowerLoom knowledge bases.

1.3 Learning and Partial Inference

Most commonly, new knowledge bases are built by knowledge engineers by explicitly modeling all relevant aspects of a particular domain. Even if this process is accelerated by reusing some already existing knowledge bases, it still requires the knowledge engineer to have a sufficient understanding of the domain to be able to construct a

knowledge base that is correct and useful. However, this understanding of the domain is not always easy to come by, in particular, if the domain knowledge is expertise and skill experts acquired over the course of their career.

While experts often have difficulty articulating how the domain should be modeled and what exactly they base their decisions on, they are usually very good in providing examples of relevant scenarios. To be able to exploit knowledge provided in the form of examples, we developed the KILTER tool set. KILTER is built on top of PowerLoom and provides a set of tools such as a neural-network-based learner for weights and weight combination over logic rules, a rule induction engine and a case-based reasoner that can each be used to exploit knowledge provided in the form of examples or cases. A central ingredient of all these tools is a special *partial inference* engine built on top of PowerLoom's backward chaining reasoner that can derive partial answers to a query even if not all necessary supporting knowledge is available. This is an important functionality required by KILTER, since matching of examples and cases is usually inexact which is not supported by standard, strict logical inference. An application of the KILTER case-based reasoner to determine decisive points in a military course of action is described in Section 3.

1.4 Large-scale Persistent Storage and Scalable Inference

One aspect of high performance knowledge bases is that they can be very large. It is therefore extremely important for tools operating on such knowledge bases to be *scalable*. For a KR&R system to be scalable, it has to (1) be able to efficiently store and access very large knowledge bases, and (2) be able to reason with them effectively. As part of our work in HPKB we addressed both of these issues.

To be able to efficiently store and access very large KBs, we built the PowerLoom Knowledge Pager. The Knowledge Pager uses a relational database (currently MySQL) to persistently store PowerLoom knowledge. To do that, relation definitions and assertions are specially encoded and then stored in the database. The current encoding scheme stores all PowerLoom assertions in a single table which is different from standard database practice, where each relation is stored in a different table. However, our

encoding scheme allows us to very easily ask standard KR&R-style queries such as “what are all facts known about Fred” which would be very difficult to ask in a standard scheme using many individual relation tables.

A newly developed indexing and access scheme then allows PowerLoom to dynamically find and page in knowledge from the database store. This scheme is lazy and does not page in any knowledge into main memory until it is actually needed. A special prefetching mechanism adds a little bit of “eagerness” to this lazy scheme in order to be able to exploit locality on the database end by paging in related (and hopefully relevant) information in a single access. Paged-in knowledge is stored in a size-limited cache, which provides speedy access to repeatedly needed information but also ensures that we won’t run out of memory even if we work with very large knowledge bases. Modifications to the knowledge base are written back out to the database, which gives us persistence. Variations of this knowledge paging mechanism have now also been applied in other DARPA programs such as RKF and EELD.

The second aspect of scalability is effective inference. PowerLoom uses a fully expressive language (a variation of first order logic) as its representation language. Performing logical inference with first order logic is inherently computationally intractable, but PowerLoom uses a variety of mechanisms such as resource bounded inference and specialized reasoners to cope with the computational complexity of logical inference.

In the context of very large knowledge bases, however, there is a second complexity aspect that needs to be attended to: inference processes that are “eager” in nature and always look at every item in the knowledge base are problematic, because it might take a very long time for them to complete. One such affected inference module is the PowerLoom description classifier which had to be designed specially to be able to work with very large knowledge bases.

Description classifiers are uniquely suited to the task of organizing conceptual networks into semantic hierarchies, and for validating the consistency of a knowledge base. Classifier inferences help to make explicit semantic relationships and derived facts that exist implicitly in a knowledge base, thereby assisting users in visualizing the contents

and consequences of a complex base of axioms and facts. Classifiers are also able to detect contradictory definitions, thereby providing a semantic check on the integrity of a domain model.

Traditionally, all classifiers have been designed to operate in a sort of “batch” mode—for every concept and instance entered into a classifier-based KR&R system the classifier computes subsumption relationships derivable between it and all other concepts. The computational overhead of classification becomes prohibitively large as knowledge bases increase in size. In practice, this prevents a classification-based KR&R system from managing very large knowledge bases; in other words, it is not scalable.

For PowerLoom, we invented a new mode of classification wherein any portion of a knowledge base can be loaded into PowerLoom’s working memory, and the PowerLoom classifier will classify only the main memory-resident knowledge entities. A special-purpose form of this scheme is PowerLoom’s module-based classification that classifies all relations and instances in a particular module only, thereby leaving large portions of the knowledge base residing in other modules untouched. This allows a user to classify the portion of the knowledge base or ontology s/he is actually interested in, without having to pay the computational overhead for classifying large amounts of knowledge that is irrelevant to the users task.

1.5 Other Tools

As programming languages drift in and out of fashion, large software systems built in those languages can become obsolete, because the expense of porting them to newer languages can be prohibitive. This phenomenon is currently afflicting systems built in Common Lisp—large research systems such as Loom are gradually becoming less useful to certain classes of users as those users migrate to other languages (e.g., to C, C++ or Java). Prior to HPKB our group developed a unique programming language called STELLA, tailored for programming intelligent symbol processing applications, that eliminates this problem. Programs written in STELLA can be translated into efficient C++ and Common Lisp programs. As part of our work in HPKB we additionally developed a STELLA-to-Java translator. A system programmed in STELLA can therefore be used by the (still considerable) body of researchers that use Common Lisp,

as well as by (more product-oriented) users who base their software on C, C++, or Java. PowerLoom is written in STELLA, and hence runs efficiently in all three different languages. As mentioned above, the OntoSaurus browser is also written in STELLA, making it highly portable as well. The use of STELLA-based technology means that research systems like PowerLoom can get transitioned much more rapidly and smoothly into commercial or product environments.

2 The OntoMorph Translator for Symbolic Knowledge

2.1 Introduction

A common problem during the life cycle of knowledge-based systems is that symbolically represented knowledge needs to be translated into some different form. As a tool to support such translation needs, we developed the OntoMorph system. OntoMorph provides a powerful rule language to represent complex syntactic transformations, and it is fully integrated with the PowerLoom KR system to allow transformations based on any mixture of syntactic and semantic criteria

For example, integration of independently developed knowledge-based components [Cohen et al., 1998], merging of overlapping ontologies [Valente et al., 1999], communication between distributed, heterogeneous agents, or porting of knowledge-based systems to use a different knowledge representation infrastructure commonly require translation, since every encoding of knowledge is based on a multitude of representational choices and assumptions. Translation needs go well beyond syntactic transformations and occur along many dimensions, such as expressivity of representation languages, modeling conventions, model coverage and granularity, representation paradigms, inference system bias, etc., and any combination thereof.

Traditionally, such translations are either performed manually via text or knowledge base editors or via special-purpose translation software. Manual translation is slow, tedious, error-prone, hard to repeat and simply not practical for certain applications. Special-purpose translation software is difficult to write, hard to maintain and not easily reusable.

Being confronted with translation problems on a frequent basis, we developed the OntoMorph system to facilitate ontology merging and the rapid generation of knowledge base (KB) translators. OntoMorph combines two powerful mechanisms to describe KB transformations: (1) *syntactic rewriting* via pattern-directed rewrite rules that allow the concise specification of sentence-level transformations based on pattern matching, and (2) *semantic rewriting* which modulates syntactic rewriting via (partial) semantic models and logical inference supported by an integrated KR system. The integration of these

mechanisms allows transformations to be based on any mixture of syntactic and semantic criteria, which is essential to support the translation needs enumerated above. The OntoMorph architecture facilitates incremental development and scripted replay of transformations, which is particularly important during KB merging operations.

2.2 The Translation Problem

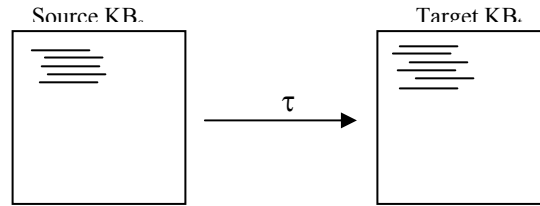


Figure 2: The knowledge translation problem.

The general problem we set out to solve is shown in Figure 2. Given some source knowledge base KB_s we want to design a transformation function τ to transform it into a target knowledge base KB_t . A fundamental assumption in this formulation is that source and target KBs are describable by a set of sentences in some linear, textual notation, where *sentence* means some independent syntactic unit as opposed to a well-formed logical formula associated with a truth value. This does not exclude graphical languages such as, for example, SNePS [Shapiro & Rapaport, 1992] or Conceptual Graphs [Sowa, 1992], since they usually also have some linear syntax to textually describe their networks. The translation does not necessarily have to span a whole knowledge base. In some cases, it might only involve single expressions.

A common correctness criterion for translation systems is that they preserve semantics, i.e., the meaning of the source and the translation has to be the same. This is not necessarily desirable for our transformation function τ , since it should be perfectly admissible to perform abstractions or semantic shifts as part of the translation. For example, one might want to map an ontology about automobiles onto an ontology of documents describing these automobiles. Since this is different from translation in the usual sense, we prefer to use the term *knowledge transformation* or *morphing*.

2.3 Dimensions of Mismatch

Despite the fact that the function τ might perform arbitrary semantic shifts, the most common scenario is to translate between different models of the same general domain. Unfortunately, these models can and in practice do differ along a multitude of dimensions. The most commonly encountered mismatches are outlined below.

2.3.1.1 KR language syntax:

Every KR language comes with its own syntax, which is probably the most mundane but nevertheless annoying mismatch. For example, here are three different ways of defining automobiles as a subclass of road vehicles, one for Loom [MacGregor, 1991], a KL-ONE-style description logic, one for MELD, the representation language used by CYC [Lenat, 1995] and one for KIF [Genesereth, 1991]:

```
Loom:  (defconcept Automobile
        "The class of passenger cars."
        :is-primitive Road-Vehicle)

MELD:  (#$isa #$Automobile #$Collection)
        (#$genls #$Automobile #$RoadVehicle)
        (#$comment #$Automobile
         "The class of passenger cars.")

KIF:   (defrelation Automobile (?x)
        "The class of passenger cars."
        :=> (Road-Vehicle ?x))
```

Apart from different surface syntax, there are also different *syntactic conventions* such as the spelling of names that are really part of the culture of the language users. For example, CYC names are mixed-case without hyphens as opposed to the hyphenated, case-insensitive spelling usually used with the other languages.

2.3.1.2 KR language expressivity:

Every KR language trades off representational expressiveness with computational tractability. For example, negation, quantification, defaults, modal operators, representation of sets, etc. are supported by some languages and not by others. When translating between languages of different expressiveness, difficult choices have to be made in how to map certain representational idioms. For example, to represent that the

typical capacity of a passenger car is five, we could use the following representation in Loom:

```
(defconcept Automobile
  :is-primitive Road-Vehicle
  :defaults (:filled-by passenger-capacity 5))
```

To represent the same in ANSI KIF which does not support defaults, one would have to resort to something like the following and then leave it up to some extra-logical means to properly reason with typicality assertions:

```
(defrelation Automobile (?x)
  :=> (and (Road-Vehicle ?x)
            (typical-passenger-capacity ?x 5)))
```

2.3.1.3 Modeling conventions:

Even if the KR language and system for source and target KB are the same, differences occur because of the way a particular domain is modeled. For example, a choice one often has to make is whether to model a certain distinction by introducing a separate class, or by introducing a qualifying attribute relation. E.g., to distinguish between tracked and wheeled vehicles, one could either introduce two subclasses of `Vehicle` called `Tracked-Vehicle` and `Wheeled-Vehicle`, or use an attribute relation as in `(traction-type My-Car wheeled)`. Which representation to choose is in most cases just a matter of taste or convention.

2.3.1.4 Model coverage and granularity:

Models differ in their coverage of a particular domain and the granularity with which distinctions are made. This is often the very reason why ontologies are merged. For example, one ontology might model cars but not trucks. Another one might represent trucks but only classify them into a few categories, while a third one might make very fine-grained distinctions between types of trucks based on their general physical structure, weight, purpose, etc.

2.3.1.5 Representation paradigms:

Different paradigms are used to represent concepts such as time, action, plans, causality, propositional attitudes, etc. For example, one model might use temporal representations

based on Allen's interval logic [Allen, 1984], while another might use a representation based on time points. Section 2.5 describes a situation where two different representations of "purpose" had to be reconciled with the help of OntoMorph.

2.3.1.6 Inference system bias:

Last but not least, another reason why models often look a certain way is that they were constructed to produce desired inferences with a particular inference engine or theorem prover. For example, in a description logic such as Loom, certain inferences are well-supported by the classifier, while others are only supported at the instance or individual level. This trade-off can influence one's choice whether to model something as a class or as an individual. See [Valente et al., 1999] for a discussion of modeling examples exhibiting inferencing bias.

2.4 OntoMorph

To facilitate the rapid specification of KB transformation functions such as τ described above, OntoMorph combines two powerful mechanisms: (1) *syntactic rewriting* via pattern-directed rewrite rules that allow the concise specification of sentence-level transformations based on pattern matching, and (2) *semantic rewriting* which modulates syntactic rewriting via (partial) semantic models and logical inference.

2.4.1 Syntactic Rewriting

To allow translation between arbitrary KR languages that can differ widely in their syntax, expressiveness, and underlying knowledge model, OntoMorph uses *syntactic rewriting* as its core mechanism. Input expressions are first tokenized into lexemes and then represented as syntax trees whose subtrees represent parenthesized groups (similar to Lisp s-expressions). The tree structure exists only logically; a tree is represented internally as a flat sequence of tokens.

For example, the expression $f(g([x], y))$ would be represented by the token sequence

``f' `(' `g' `(' `[' `x' `]' `,' `y' `)' `)'``

which, logically, represents the syntax tree shown in Figure 3. The significance of the

tree structure is that complete subtrees can be matched by a single pattern variable, and that sequence variables do not consume tokens beyond subtree boundaries.

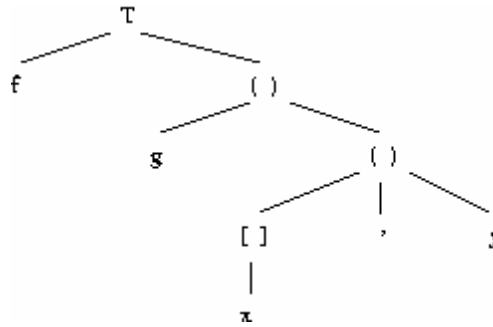


Figure 3: **Syntax tree representation of $f(g([x], y))$**

OntoMorph's syntactic rewrite rules have this general form:

$$pattern ==> result$$

The left-hand-side pattern matches and destructures one or more syntax trees while the right-hand side generates new trees of the desired format by explicitly specifying new structure, reassembling some of the destructured information and by possibly further rewriting some subexpressions. For example, a very simple rule to convert a MELD type assertion into its Loom analogue would look like this (pattern variables are prefixed with a '?'):

```
(isa ?x ?class) ==> (tell (?class ?x))
```

The ability to describe such transformations in a very direct and concise fashion was an important design objective for OntoMorph. When researching the relevant parsing and pattern-match literature and technology, we found that a language called Plisp (or Pattern Lisp) [Smith, 1990], which in turn is a direct descendent of the Lisp 70 pattern matcher [Tesler et al., 1973], came closest to our intuitions on how such transformations should be represented and executed. Unfortunately, none of these systems is alive and well anymore, so we had to develop our own version.

2.4.1.1 Pattern Language

OntoMorph's pattern language and execution model is strongly influenced by Plisp, even though the actual surface syntax is quite different. The pattern language can match and

destructure arbitrarily nested syntax trees in a direct and concise fashion. A short overview of the available constructs is given below:

Literals such as `foo`, `"bar"`, `42`, `(`, `(a (b c))`, etc., have to be matched by identical literal tokens (or token sequences).

Variables (indicated by a `?`-prefix), e.g., `?x`, `?why` or the anonymous variable `?`, can match individual input tokens such as `foo` or a token sequence representing a tree such as `(a (b c))`. Once a variable is bound, it can only be matched by literal tokens matching its binding.

Sequence variables (indicated by a `??`-prefix), e.g., `??h`, `??tail` or the anonymous variable `??`, can match tree subsequences such as `c (d)` in the tree `(a b c (d))`. For example, the pattern `(??x b ??y)` matches the tree `(a b c (d))` by binding `??x` to the single-element sequence `a` and `??y` to the sequence `c (d)`. Sequence variables cannot consume tokens beyond subtree boundaries.

Grouping (expressed via braces) defines compound patterns. For example, the pattern `{a ?x c}` can match the token sequence `a b c`. Groups are also used to apply pattern modifiers such as repetition to compound patterns.

Alternatives (expressed via vertical bars) define disjunctive patterns such as `{a | (b ?x) | c d}`. The pattern matches if one of its components succeeds. *Optionals* such as `{a b [c]}` are syntactic sugar for the more verbose `{a b | a b c}` notation.

Repetition (expressed with the usual `*` or `+` notation) indicates that a pattern can be matched multiple times. As a generalization, an `m-n` range can be supplied to mandate that there have to be at least `m` and at most `n` matches. For example, `{a | b}+` matches any sequence of `a`'s and `b`'s with length ≥ 1 , `{a | b}*1-2` matches only those sequences with lengths between 1 and 2.

Input binding binds the input matched by a complex pattern to a single variable. This is useful if a pattern has alternatives and it is necessary to refer to what was actually matched by it in the right-hand side of a rewrite rule (without alternatives, the same could

be achieved by literally repeating the pattern). For example, `?x := {a | (b ?y) | c}` matched against `(b d)` binds `?x` to `(b d)`.

Below is an example pattern that combines various of the elements described above to match and destructure a Loom concept definition (note, that the example only covers some aspects of the Loom concept language). The alternatives in combination with the repetition construct allow the keyword/value pairs to appear in any order. The construction for the `:annotations` keyword extracts a documentation string (which might appear in various ways) while ignoring everything else:

```
(defconcept ?name
  {?is := {:is | :is-primitive} ?def |
   :characteristic ?c |
   :annotations
   ?a := {(documentation ?d) |
           (:and ?? (documentation ?d) ??) |
           ?}}*0-3)
```

The pattern matches and destructures concept definitions such as this one:

```
(defconcept Dog
  :annotations
    (:and Class (documentation "Canine"))
  :is-primitive Animal)
```

2.4.1.2 Execution Model

Rewrite rules are applied according to the following simple execution model: Initially, an input stream is constructed consisting of the token sequence representing the input expression. When a rewrite rule is applied, its left-hand-side pattern consumes tokens from the input stream by matching them against the elements of the pattern. If the pattern succeeds, the right-hand-side result is assembled and the resulting tokens are pushed back onto the input stream where they replace the consumed input and become available as input to further rewrite rules. For example, assume we have the following input stream:

```
`(' `isa' `car1' `Ford' `)' `(' ...
```

Now we apply the type transformation rule from before:

```
(isa ?x ?c) ==> (tell (?c ?x))
```

Applying the rule modifies the input stream to the following:

```
`(' `tell' `(' `Ford' `car1' `)' `)' `(' ...
```

Assembly of a rule result involves collecting its right-hand-side component tokens from left to right into a temporary store. Literal tokens such as the `tell` above are simply copied, variables are substituted by their bindings, and functions and recursive rule invocations (explained below) are evaluated and their results collected. Once all right-hand-side components have been successfully evaluated, the content of the temporary store is prepended to the input stream where it replaces the input consumed by the left-hand-side.

Rewrite rules are always assembled into rule sets of the following form:

```
(defruleset name
  pattern1 ==> result1
  ...
  patternn ==> resultn)
```

The individual rules are implicitly OR-ed and tried in sequence. The ruleset succeeds with the result of the first successful element rule.

Explicit invocation of named rulesets is the primary mechanism to achieve recursion, which is necessary to handle the translation of recursive structures. Apart from this computational aspect, grouping rules improves modularity, and it also greatly improves efficiency, since it restricts the set of rules tried to rewrite any given subexpression.

While matching a pattern and also during the assembly of the right-hand-side result which might involve further rewrites, a rule may fail. In that case execution backtracks to the most recent match choice point. After all input has been consumed and no more rules need to be applied, the process terminates and the resulting state of the input stream constitutes the result of the rewrite operation which is then either printed to some storage medium or used directly as part of a KB operation such as assertion or retrieval.

2.4.1.3 Function Calls and Rule Invocations

To allow the parsing and rewriting of recursive structures, other rulesets as well as built-in functions can be invoked explicitly anywhere in a pattern. Such invocations are written with an angle bracket syntax to distinguish them from the regular syntax tree notation. For example, the call `<Term ?x>` invokes a function or ruleset called `Term` on the argument `?x`. Before the function is called, its arguments are evaluated and the results pushed back onto the input stream from which they are then consumed. Excess or missing arguments are left on or filled in from the remainder of the input. When a function or ruleset invocation on the left-hand side of a rule returns, its result gets pushed back onto the input where it immediately becomes available to subsequent pattern elements. On the right-hand side (as described above), the result gets first collected in a temporary store until all right-hand-side tokens of the rule have been evaluated.

The following two rule sets constitute a simple transformation system for arithmetic expressions (note, that the `+` and `*` symbols need to be escaped to treat them as ordinary characters):

```
(defruleset Term
  (?op := { \+ | - | \* | / } ?x ?y)
  ==> (?op <Term ?x> <Term ?y>)
  (1\+ ?x) ==> (\+ <Term ?x> 1)
  (1- ?x) ==> (- <Term ?x> 1)
  (square ?x) ==> (\* <Term ?x> <Term ?x>)
  ?x ==> ?x)

(defruleset Condition
  (lt ?x ?y)
  ==> (negative? (- <Term ?x> <Term ?y>))
  (gt ?x ?y) ==> <Condition (lt ?y ?x)>)
```

To apply these rules, we can use the `OntoMorph` function `rewrite` which takes an input expression and a start rule as arguments. For example,

```
(rewrite (gt (/ (1+ M) N) (square N))
         Condition)
```

returns the following result:

```
(negative? (- (* N N) (/ (+ M 1) N)))
```

Currently, OntoMorph uses a Lisp-style reader to tokenize the input into individual lexemes. Future versions will allow the specification of customized tokenizers in order to support the translation of languages with different lexical conventions.

2.4.2 Semantic Rewriting

Syntactic rewriting is a powerful mechanism to describe pattern-based, sentence-level transformations. However, it is not sufficient if the transformations have to consider a larger portion of the source KB, possibly requiring logical inference. A simple example of such a transformation is conflation. Suppose one wants to conflate all subclasses of `Truck` occurring in some ontology about vehicles into a single `Truck` class. This involves among other things the rewriting of all type assertions involving trucks. Using syntactic rewriting alone, one would need a rule such as the following that explicitly lists all subtypes of `Truck`:

```
(defruleset Conflate-Truck-Types
  ({Light-Truck | Heavy-Truck | ...} ?x)
  ==> (Truck ?x))
```

For large taxonomies this is of course neither elegant nor feasible. Instead of the purely syntactic test based on truck class names, a semantic test is needed to check whether a particular class is a subclass of `Truck`.

To facilitate the utilization of semantic information, OntoMorph is built on top of the PowerLoom knowledge representation system. PowerLoom is a successor to the Loom system that supports definitions and rules in a typed variant of KIF combined with a powerful inference engine and a classifier. Wherever a function call is legal in a rewrite rule, a PowerLoom function can be called to change or access the state of the current KB.

One way to solve the conflation problem is to establish a partial mirror of the source KB within an intermediate PowerLoom KB. This can be done with a specialized set of rewrite rules that import source sentences representing taxonomic relationships, but ignoring all other information, for example, by only paying attention to `subset` and `superset` assertions. This step can be viewed as the first pass of a two-pass translation scheme. In the second pass, the actual translation rules are applied, but now they can also

access the semantic information established in the first pass. Making use of the imported taxonomic knowledge, the following rule can conflate all truck types:

```
(defruleset Conflate-Truck-Types
  {(?class ?x) <ask (subset-of ?class Truck)>}
  ==> (Truck ?x))
```

The left-hand-side contains a group of patterns which is treated as a conjunction. The first conjunct `(?class ?x)` simply matches any type assertion. The second one calls `ask` which triggers a PowerLoom query. Note that `?class` will be substituted with the matched class name, thus, the query will be fully ground. Since `ask` is a boolean-valued function, its result will simply be treated as a test instead of being pushed back onto the input stream.

Using semantic import rules, an arbitrarily precise image of the source KB semantics can be established within PowerLoom (limited only by the expressiveness of first-order logic). Then syntactic rewrite rules can use the imported semantic information to perform rewrites based on any mixture of syntactic and semantic criteria.

Obviously, the precision of the semantic import will affect the quality of the translation. For example, in the scenario above the semantic import only considered `subset` and `superset` assertions. Depending on the nature of the source KB, there might be other information and rules that would allow one to infer additional taxonomic relationships. These would then not be inferable within the partial PowerLoom mirror KB which might adversely affect the translation quality.

Whether this is a problem and how to best solve it has to be decided on a case-by-case basis. One solution is to use PowerLoom as an interlingua and import everything from the source KB (again, this is limited only by the expressiveness of PowerLoom). The disadvantage of this scheme is that one effectively needs two sets of translation rules, one to translate from the source into PowerLoom, and one to go from PowerLoom to the target representation. Alternatively, it might be possible to call out to the KR system that has the source KB loaded and use its inferencing capabilities directly. This can either be done via some special-purpose API, or, if supported, via a protocol such as OKBC [Chaudhri et al., 1998]. Which route to take will depend on a variety of pragmatic

factors. For the OntoMorph applications constructed to date, importing partial semantic information into PowerLoom was sufficient to support all rewriting needs.

2.5 OntoMorph Applications

OntoMorph has already been successfully applied in a couple of domains. One involved the translation of military scenario information for a plan critiquing system. In the second it formed the core of an agent translation service called Rosetta, where it was used to translate messages between two communicating planning agents that used different representations for goals.

2.5.1 Course of Action Critiquer

One of the challenge problems that drove the second phase of DARPA's High Performance Knowledge Bases (HPKB) project [Cohen et al. 1998] was to develop critiquing systems for military courses of actions (or COAs) which are high-level, plan-like descriptions of military operations. To represent a particular COA, scenario information from a graphical sketch pad was fused with information from a natural language description of the COA by a program called the Fusion engine. The combined description of the COA was represented in CYC's MELD language and then fed to five independent critiquing systems built by different teams. Only one of the critiquers was using CYC directly and did not have to translate the Fusion engine output. All others had to use some form of translation system. Many different scenarios had to be handled in a tight evaluation schedule, thus, manual translation was not an option. OntoMorph was chosen to translate the Fusion output for the critiquer based on the EXPECT knowledge acquisition system [Gil, 1994] which uses Loom to represent its knowledge. What follows is a list of translation issues that arose, and how they were solved:

2.5.1.1 Different Names:

While most of the names generated by the Fusion engine were shared by the EXPECT critiquer, some of them differed due to parallel independent development of critiquers and ontologies as well as personal style. Renaming was taken care of with simple rules like the following:

```
(DEFRULESET rename-collection
```

```

Fix-MilitaryTask ==> FIX
{ProtectingSomething |
  ProtectingPhysicalRegion} ==> PROTECT
Translation-LocationChange ==> MOVE
...)

```

2.5.1.2 Different Syntax:

OntoMorph started with a KIF translation of the Fusion output which still contained various MELD idioms that needed to be translated into Loom syntax. For example, `isa` assertions such as `(isa task1 Fix-MilitaryTask)` had to be translated into the Loom idiom `(FIX task1)` (which here also involved a name change). MELD frame predicates were also easily translated into Loom with the following rule:

```

(DEFRULESET rewrite-frame-predicate
  (relationInstanceExistsCount
    ?relation ?instance ?type ?count)
  ==> (:ABOUT <rewrite-term ?instance>
    (:EXACTLY ?count
      <rename-relation ?relation>
      <rename-collection ?type>)))

```

2.5.1.3 Different Representations:

The most challenging difference to overcome was the different representations used to represent the purposes of tasks. The Fusion engine used an idiom that related a task with a proposition whose truth was supposed to be brought about by carrying out the task. For example, to state that the purpose of the task carried out by `BlueDivision1` was to protect `Boundary1`, the following representation was used:

```

(taskHasPurpose BlueDivisionTask
  (thereExists ?p
    (isa ?p
      (CollectionSubsetFn
        ProtectingSomething
        (TheSetOf ?obj
          (and (objectTakenCareOf
            ?obj Boundary1)
            (performedBy
              ?obj BlueDivision1)))))))

```

This can roughly be paraphrased as follows: The purpose of `BlueDivisionTask` is to bring about the existence of an event `?p` that is an instance of the event type `ProtectingSomething` restricted by the set of events in which `BlueDivision1`

takes care of Boundary1 (the restriction is expressed via the `CollectionSubsetFn` construction). This representation goes far beyond the expressiveness of Loom which does not have a way to represent higher-order sentences such as the above. It also did not meet the requirements of the EXPECT critiquer, which needed a reified purpose representation such as the following:

```
(AND (PROTECT protect00)
      (PURPOSE-ACTION protect00)
      (PURPOSE-OF BlueDivisionTask protect00)
      (ACTION-OBJ protect00 Boundary1)
      (WHO protect00 BlueDivision1))
```

The final version of the Fusion engine only used three structurally different purpose representation patterns. Each of them could be handled by an OntoMorph rule such as the following:

```
(DEFRULESET rewrite-purpose-pattern1
  {(taskHasPurpose ?task
    (thereExists ?var
      (isa ?var
        (CollectionSubsetFn
          ?type
          (TheSetOf ?action ?body))))))
  <generate-unique-name
  <rename-collection ?type>>
  ?purpose}
==> (AND
      (<rename-collection ?type> ?purpose)
      (PURPOSE-ACTION ?purpose)
      (PURPOSE-OF ?task ?purpose)
      <rewrite-purpose-setof-body
      ?body ?action ?purpose>))
```

The mapping between the two representations is very direct and makes good use of OntoMorph's destructuring facilities for syntax trees. The only complication is the extra right-hand-side function call to create a skolem individual needed to represent the reified purpose. This is taken care of by a call to the built-in function `generate-unique-name` which bases the generated name on the supplied argument (in this case, the renamed base event type). It does not consume anything from the input stream but simply pushes the result back onto it where it is then consumed by the `?purpose` variable.

2.5.1.4 Missing Representations:

Some information needed by the EXPECT critiquer such as COA substructure and task/subCOA associations was not explicitly represented and needed to be recovered by some of OntoMorph's semantic rewrite features, e.g., by keying in on "meta-information" such as where in the Fusion output certain assertions were made.

For example, to associate a task with a particular subCOA, it was necessary to track what tasks were performed by what unit which was handled by the following two rules:

```
(DEFRULESET track-COA-assertion
  (unitAssignedToTask ?task ?unit)
  ==> <!ASSERT
      (AND (Term ?task) (Term ?unit)
        (unitAssignedToTask
          ?task ?unit))>)

(DEFRULESET get-task-assigned-to-unit
  {?unit
   <@RETRIEVE \?t
    (= (unitAssignedToTask \?t) ?unit)>
   ?task}
  ==> <OBJECT-NAME ?task>)
```

The first rule creates a PowerLoom assertion for each `unitAssignedToTask` statement in the Fusion scenario. PowerLoom expects all its objects to be typed before they are used which is the reason for the additional `Term` assertions. The second rule retrieves the task recorded for a particular unit which was then used to associate it with the sub-COA in which the particular unit was involved. Note, that the `?t` variable within the PowerLoom `retrieve` statement is escaped, since it is a retrieval variable and not a pattern variable of the rewrite rule. The `?unit` variable, however, is a pattern variable, thus, its binding is substituted before the retrieval is executed and is seen by PowerLoom as an ordinary constant.

The complete translator was comprised of about 30 rulesets, 10 of which were necessary just to track unrepresented COA structure. The size of the translator was about 15 kilobytes of text.

2.5.2 Rosetta Agent Translation Service

Rosetta is a prototype of an ontology-based translation service operating in a domain of planning agents. It reengineers some aspects of a technology integration effort described in [Cox & Veloso, 1997] which connects the ForMAT case-based planning tool [Mulvehill & Christey, 1995] with the Prodigy/Analogy planner [Veloso, 1994; Veloso et al. 1995]. In the original experiment, special-purpose translators were constructed to allow ForMAT and Prodigy/Analogy to communicate. Rosetta is an attempt to show how these translators can be replaced with a more flexible, general-purpose translation architecture that promotes reuse and that can scale up to large numbers of heterogeneous, communicating agents. The full motivation and details of the Rosetta architecture are given in [Blythe et al., 2000]. Here we will only touch on some aspects and how they are handled by the OntoMorph system.

The main idea behind Rosetta is that it provides a representation interlingua in conjunction with a repository of broad-coverage as well as domain-specific ontologies that can be used to represent content expressions exchanged by heterogeneous communicating agents. Each agent is associated with a wrapper that (1) translates its message content language into the interlingua used by Rosetta, and (2) if necessary, aligns terms of the agent ontology with Rosetta's ontologies. Within Rosetta, each agent is associated with a model that represents relevant aspects of the agent's domain. As motivated in Section 2.3, using the same KR language and system to model a domain does not by itself eliminate the need for translation, since different representations can be used to express the same semantic content. To facilitate translations between such different representations, Rosetta has a library of representation reformulation rules.

To translate a message between agents A and B, agent A first uses its wrapper to translate the message content into Rosetta's format and sends it to Rosetta. Rosetta then checks whether any reformulation rules need to be applied to make the message understandable by agent B, and, if so, applies them. The resulting message is then sent to agent B which uses its own wrapper to translate it into its internal format. One of the advantages of this architecture is that the portion of the necessary translation mappings encodable in the wrappers grows only linearly with the number of different agent classes.

The uses of OntoMorph within this scenario were twofold: (1) It provided an obvious solution to implement the agent wrappers by primarily relying on its syntactic rewriting features. (2) Its semantic rewriting features were used to implement the necessary representation reformulation rules. For example, the following top-level rule was used to translate a goal posted by the ForMAT planning tool into the Rosetta representation (note, that only the most relevant aspects of these rules are reproduced to save space):

```
(defruleset format-to-rosetta-wrapper
  {(:goal ?goal)
   <translate-format-goal-to-rosetta ?goal>
   ?translated-goal}
 ==>
  (message
   (content
    (find (object plans)
          (for (Objective-Based-Goal
                ?translated-goal))))
    ...)) ...)
```

This rule translates a ForMAT request such as

```
(:goal
 (G-144 :Send-Hawk
        ((force 42nd-Batt)
         (geographic-location Big-Town))))
```

into the following representation understandable by Rosetta (the message content language used for this prototype is based on the verb clause goal language used by the EXPECT system):

```
(message
 (content
  (find (object plans)
        (for (Objective-Based-Goal
              (send-unit
               (object 42nd-Batt)
               (to Big-Town))))))
  ...))
```

Once this message arrives at Rosetta, it is handled by its top-level translation rule whose main purpose it is to trigger the translation of content expressions:

```
(defruleset translate-rosetta-message
  {(message
    {(content ?content) |
```

```

    ...}*4-4)
    <map-performative ?content>
    ?mapped-performative}
==>
(message
  (content ?mapped-performative)
  ...) ...)

```

One of the interesting aspects of the communication between ForMAT and Prodigy/Analogy is that ForMAT uses an objective-based or verb-centered representation such as "send troops to X" to represent its goals. Prodigy/Analogy, on the other hand, needs to be given a state-based goal representation such as "troops deployed at X" to generate a plan. To be able to represent these different kinds of goals as well as other planning-related aspects, Rosetta employed the PLANET ontology developed by Blythe and Gil [Blythe et al., 2000]. To translate between objective-based and state-based goals, Rosetta uses a (heuristic) reformulation rule that looks for the primary effect of the planning operator describing the objective-based goal to serve as its state-based translation. Here are two of the central reformulation rules involved in this mapping:

```

(defruleset map-objective-to-state-based-goal
  {?goal-instance ??roles
   <find-equivalent-operator ?goal-instance>
   ?operator
   <get-primary-effect ?operator> ?effect}
==>
  (State-Based-Goal
   <map-operator-and-roles
    ?operator ?effect (??roles)>))

(defruleset find-equivalent-operator
  {?goal-instance
   @most-specific-named-descriptions
   <retrieve-tuples all \?op
    (and (member-of ?goal-instance \?op)
         (context-of
          \?op <get-agent-model
            <current-receiver>>))
    (exists \?effect
      (role-type primary-effects
        \?op \?effect)))>>
   ?equiv-operator}
==>
  <object-name ?equiv-operator>)

```

The first thing Rosetta does is to find a planning operator in its model of the

Prodigy/Analogy agent that is a suitable match for the operator requested by ForMAT. It does so by looking for the most specific operator description that matches the description of the goal posted by ForMAT by using a PowerLoom subsumption test. After the operator is found, its primary effect is used as a state-based goal description that can be passed on to Prodigy. Once the top-level `translate-rosetta-message` rule terminates, the translated message looks like this and is sent to Prodigy:

```
(message
  (content
    (find (object plans)
      (for (State-Based-Goal
        (is-deployed
          (object 42nd-Batt)
          (at Big-Town))))))
  ...)
```

Finally, the Prodigy/Analogy wrapper translates that into the following, which can be sent directly to the planner:

```
(:find-plans
  (is-deployed 42nd-Batt Big-Town))
```

The Rosetta application provides a nice testbed for all aspects of OntoMorph. Syntactic rewriting is exercised in the agent wrappers, semantic rewriting is exercised to perform representation reformulations, and a mixture of both controls the scripting of the overall translation process. Furthermore, the tight integration with the PowerLoom KR system and the interpreted nature of the rewrite rules provide for a very productive, incremental development cycle.

2.6 Related Work

Ontolingua [Gruber, 1993; Fikes et al., 1997] is an attempt to avoid the translation problem by providing a centralized ontology repository that encourages reuse, and an ontology specification language that serves as an interlingua whose representational primitives can be translated into a variety of target KR languages by special-purpose translators. However, since the generated translations cannot be controlled, modifications such as changing modeling conventions or performing semantic shifts is not possible. While avoiding translation is always a good strategy, it is not always

possible such as in the case of distributed, heterogeneous agents. Using one big, centralized ontology as done by CYC has similar drawbacks. In particular, it becomes problematic when a smaller system that only relies on a portion of the ontology needs to be fielded. Another alternative to translation is the use of lifting axioms as done in [Frank et al., 1999]. Lifting axioms can only be used in systems expressive enough to support them. Another drawback is that they perform translations via logical inference at query time, which could be prohibitively expensive.

Since part of OntoMorph can be viewed as a parser specification system, it is legitimate to ask how it compares to other parsing technology such as YACC, definite clause grammars, natural language parsers such as ATNs, etc. YACC parsers are only applicable to context-free languages that are LR(1), which is too restrictive for a general-purpose translation system. Natural language parsers such as ATNs could in principal be used to implement a rewrite system, but since they are geared towards parsing of natural language sentences instead of arbitrary syntax trees, the specification would be less direct and more difficult. Definite clause grammars probably come closest to our desiderata for direct and concise specification of transformation rules, however, extra support would be necessary to support certain conveniences of the OntoMorph pattern language such as sequence variables and bounded repetition of compound patterns. Additionally, the integration with a KR system such a PowerLoom would still be missing which is a crucial part of OntoMorph's utility. Similar objections hold for languages such as POP-11 which already provide some of the pattern match functionality needed by OntoMorph, but lack the combination of features and the integration with a KR system such as PowerLoom.

2.7 Future Work: Ontology Merging

One of the primary motivations for the development of OntoMorph was to support merging of overlapping ontologies. Merging two or more source ontologies into a merged ontology involves the following steps:

1. Finding semantic overlap or *hypothesizing alignments*.
2. Designing transformations to bring the sources into mutual agreement.

3. Editing or *morphing* the sources to carry out the transformations.
4. Taking the union of the morphed sources.
5. Checking the result for consistency, uniformity, and non-redundancy and if necessary repeating some or all of the steps above.

These steps have different degrees of difficulty and are supported to various degrees by the state of the art. For example, techniques for hypothesizing alignments have been developed during large-scale ontology merging tasks as described in [Knight & Luk, 1994; Hovy, 1998; McGuinness et al., 2000], and consistency checking is already fairly well supported by today's KR systems. Designing the necessary transformations is probably the most difficult and least automatable task, since it involves understanding the meaning of the representations. Additionally, this step often involves human negotiation to reconcile competing views on how a particular modeling problem should be solved.

At the center of every merging operation is step 3, since before ontologies can be merged they have to be transformed into a common format with common names, common syntax, uniform modeling assumptions, etc., which always involves some of the transformation operations described in Section 2.3. Since merging is an iterative process, it is very important that these transformations can be specified easily and carried out repeatedly and automatically with a tool such as OntoMorph. This is even more important in the context of tracking changes to one of the sources in a later re-merge. Without a clear and executable specification of the transformations used in the initial merge, much of the merging work has to be redone by hand. By using a tool such as OntoMorph, many of the necessary transformation rules will be reusable as is, and only the changed and extended portions of the modified source ontology will require adapted or new rewrite rules.

3 Determining Decisive Points through Case-Based Reasoning

3.1 Overview

The DARPA High Performance Knowledge Bases (HPKB) program has detailed a number of real military problems that challenge the current state of the art in artificial intelligence and knowledge based systems. These problems cover areas where automated solutions could greatly benefit the military, but where automation has proven difficult with traditional methods. Below we describe our solution to one of these problems: determining focus points for military planners.

One of the most difficult tasks in military planning is determining an appropriate point of focus, called a *decisive point*. A military decisive point is the point on a battlefield where a military course of action (i.e., plan) should be directed. Military experts believe that identifying effective decisive points is an art, where proficiency comes only through experience [Jones, 1999]. Experts find it difficult to verbalize their problem-solving knowledge and cannot easily teach decisive point reasoning. Consequently, it has proven difficult to automate this process with traditional expert systems technology.

Our solution avoids this knowledge acquisition bottleneck by acquiring knowledge from examples rather than expert rules. Even when experts cannot describe their reasoning, they can provide examples or case histories of decisive points. These examples can be used as a case base for a case-based reasoner or as a set of training examples for an inductive learning algorithm. In both situations, the examples provide significant knowledge content. While both case-based reasoning and inductive learning are valid solutions, this paper focuses on the case-based reasoning approach.

The decisive point problem presents two important challenges to case-based reasoning: how to manage structural case knowledge and how to fuse expert, rule-based knowledge. Most case-based reasoning applications operate on flat feature vectors and are incompatible with relational representations. The decisive point case knowledge, however, is inherently relational and does not lend itself to a feature vector

representation. The second challenge concerns existing expert knowledge. In the decisive point problem, experts cannot provide a complete solution but can provide knowledge fragments that are useful in determining decisive points. A successful case-based reasoner must leverage both rule-based and example-based knowledge.

Below we present a novel case-based solution to the decisive point problem that addresses each of these challenges. The reasoner is part of the KILTER learning toolset within the PowerLoom knowledge representation system and uses an innovative combination of nearest neighbor, graph search, neural networks, and natural deduction to build, match, and reason with relational case knowledge. Since the reasoner is implemented within PowerLoom it also exploits any existing rule-based knowledge about the problem. Such an approach can be thought of as *knowledge-rich cased-based reasoning*.

3.2 The Decisive Point Problem

3.2.1 Problem Description

A course of action (COA) is defined by the US military as a sketchy plan that describes how a military unit will carry out its mission. COAs are generated by a military planning staff through a well-specified process called the *military decision-making process*, where numerous competing COAs are developed and analyzed. All COAs have a point of focus, called the decisive point, where the military directs its combat effort. Decisive points normally refer to a feature on a map such as a geographic region or a specific military unit.¹ The most effective decisive points match a military strength against an enemy weakness.

Military planners place great emphasis on understanding and exploiting the best decisive points for a given mission. Unfortunately, military experts agree that there is no general procedure for determining effective decisive points [Jones, 1999]. Decisive points are normally chosen from a "gut feeling" rather than by following strict military doctrine. Military students become adept at recognizing good decisive points simply through trial

¹Decisive points properly refer to a physical location and a time. Here we are concerned only with the physical component.

and error. Given numerous examples of decisive points and direct feedback on whether the missions were accomplished, students begin to form an internal model of good and bad decisive points. Unfortunately, students and experts find it difficult to translate this model into a concise set of rules. A typical response when asked what make something a good decisive point is "I know it when I see it."

While experts cannot provide a complete set of rules to describe their reasoning, they can provide some knowledge fragments that are incomplete, but still useful. Some of this knowledge is domain general such as knowledge about geography and geographic relations (e.g., directions, relative distances, betweenness, etc.) and some is specific to decisive points. One example of a domain specific knowledge fragment is the fact that linear features such as rivers, borders, and phase lines are not good decisive points because they do not provide a single point of focus. An example of domain general knowledge is the fact that if x is east of y , then y is west of x . In addition to these knowledge fragments, experts can provide examples of their decisive point choices along with a measure of goodness. The goodness measure may come from an evaluation in a simulator or may be artificially generated by the expert.

The challenge problem is as follows. Given several knowledge fragments from experts and a set of decisive point cases, build a system to evaluate future decisive points. The system should input all scenario and mission related information and output a ranking of the best decisive points.

3.2.2 A Case-Based Reasoning Solution

Given the lack of existing knowledge and the availability of decisive point examples, a machine learning approach seems appropriate. Case-based reasoning is a branch of machine learning that has many pseudonyms including instance-based learning, lazy learning, and nearest neighbor. The tenet of case-based methods is that solutions to previous problems should be explicitly adapted and reused in similar future problems.

Figure 4 gives a basic case-based algorithm. Each example is explicitly stored in a case base. Once a *query* example is presented, the reasoner passes through two phases: case matching and solution adaptation. In the matching phase, the reasoner compares the

Figure 4: Case-based Reasoning High Level Algorithm. In the decisive point problem, x refers to a decisive point description and $f(x)$ refers to the goodness of the decisive point.

For each training example $(x, f(x))$, add the example to case base, C
Given a new query example x_q
 Find the k examples, $x_i \dots x_k$, in C nearest to x_q
 Return a combination of $f(x_i) \dots f(x_k)$

query example to each stored case and computes a distance measure. In the adaptation phase, the solutions from the cases with the lowest distance are combined and adapted to fit the query. Combination and adaptation are normally problem-specific procedures.

We adopted a case-approach over other machine learning methods for several reasons. First, we've found military experts to be very receptive to case-based reasoning because it often reflects their own reasoning. They understand the high level case-based algorithm and are more likely to trust its answer than, for example, a neural network, which operates more as a black box. Second, case-based reasoning provides human-understandable explanations. Along with the answer, a case-based approach can provide the relevant cases used to compute the answer. Other learning systems are not as verbose. For example, a rule induction algorithm often generates rules which do not make any sense to the user and consequently its explanations are often readable but not meaningful [Pazzani, Mani & Shankle, 1997].

The final motivation for case-based reasoning is that case-based methods do not form an explicit representation of the hypothesis and thus have almost no computational overhead at training time. Rule induction methods build a set of rules and neural network methods form a neural network. A case-based method simply stores all of the training examples, which is why it is often called lazy learning. The advantage of being lazy is that it does not have to recompute its hypothesis when new training examples are provided. It simply stores them with the others. Other learning methods have to adjust or often recompute their explicit hypothesis representations any time new training examples are provided, which creates a significant overhead expense. Of course, case-based methods have

computational overhead also, but it comes only when queries are issued, not when training examples are added. In our experience, experts are often happier to wait when the system is answering a question than when they are simply updating it.

3.2.3 Technical Hurdles

We identified two major challenges in applying case-based reasoning to the decisive point challenge problem: managing structured case knowledge and fusing rule-based and example-based knowledge. Unfortunately, most case-based applications do not address either. This section outlines these challenges and relates their importance to the decisive point problem.

3.2.3.1 Relational Case Knowledge

When evaluating a potential decisive point, experts rely on several different sources of knowledge. They consider the overall mission and its objectives, knowledge about the terrain and other geographic features, and knowledge about specific military units. Since these sources are critical in evaluating a decisive point, they should be included in the decisive point case description. In other words, a decisive point case should include all features or characteristics that lead experts to conclude that this is a good or bad decisive point.

It is difficult to imagine a non-relational representation for this type of knowledge. Consider knowledge about the terrain, which describes different geographical features such as rivers, mountains, and roads. These objects have basic properties such as position, length, and width, but the most interesting characteristics are how they are related to one another. For example, a mountain may be between two military units, and would indicate that these units are blocked from each other. This type of knowledge requires relations between specific instances and resembles a structured hypergraph where nodes reflect instances and arcs represent relations. Figure 5 gives an example of the type of relational knowledge in decisive point cases.

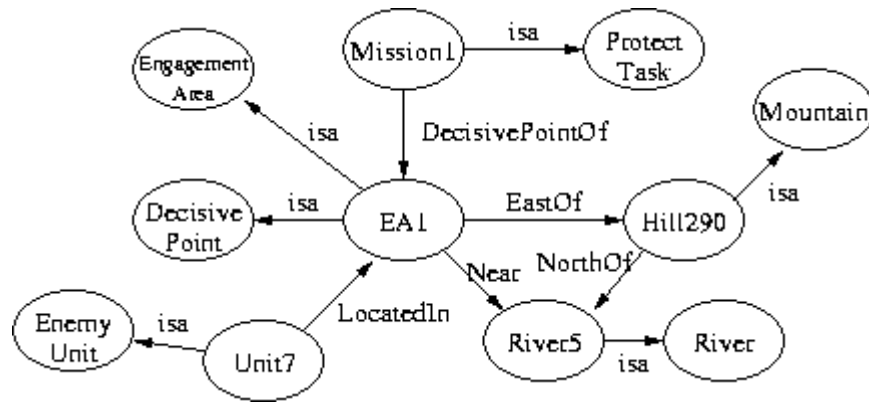


Figure 5: A partial decisive point description.

Unfortunately, almost all implementations of case-based reasoning operate on a flat set of feature values and cannot match relational cases [Gebhardt, 1997; Kolodner, 1993]. Thus, a case-based solution to the decisive point problem requires either an algorithm to translate the relational knowledge into a feature vector representation or a case-matcher that can directly compare structured cases. In Section 3.3.2, we describe a case-based system that uses both strategies.

A second problem with relational case knowledge concerns the dimensions along which two cases are compared. Traditional case-based methods assume that this information is given *a priori* (e.g., features in the feature vector). Unfortunately, when reasoning with structured knowledge we are normally not afforded this luxury. For example, decisive point cases can be described using any number of instances, properties, and relations. In other words, there is no single convention for describing a decisive point case; each is likely to have a different knowledge structure. Since we do not have a set of properties and relations common to all cases, we must devise another strategy for generating the relevant dimensions for comparison.

3.2.3.2 Combining Rule-based Knowledge with Examples

Traditional case-based reasoning methods utilize one source of knowledge: the case base. Case-based reasoning, along with most machine learning methods, normally ignore any existing rule-based knowledge about the domain. The rationale is twofold. First, in most

cases training data is plentiful and good performance can be achieved with examples alone. Second, it is not obvious how these methods could exploit such knowledge.

In the HPKB decisive point challenge problem, the training data was rather sparse. We were given only 50 cases of previously identified decisive points, which, given the complexity of the problem, represents a limited sampling of the overall problem-space distribution. Thus, to achieve higher levels of performance, one cannot rely on examples alone.

Unfortunately, little work has been done to develop methods that combine both rule-based and example-based knowledge. Typically, builders of intelligent systems fall into one of two camps. The knowledge engineering camp builds everything from expert rules. The machine learning camp learns everything from examples. This problem requires elements of both to be successful.

3.3 Knowledge-Rich Case-Based Reasoning

To overcome the challenges outlined in the previous section, we implemented a case-based reasoning module in the PowerLoom knowledge representation system. The reasoner combines nearest neighbor, neural networks, graph search, and natural deduction to reason effectively with relational, example-based knowledge and rule-based knowledge. We characterize this approach as knowledge-rich case-based reasoning.

The overall algorithm for ranking decisive points is given in Figure 6 and can be summarized as follows. Cases are stored in PowerLoom as a set of relational facts. Criteria for comparing two cases are generated from the case base using a graph search algorithm called GSA. PowerLoom then maps each case into the criteria and constructs *match vectors*. The match vectors are fed into a neural network which is trained to build a compact, semantically-rich representation for each case. The semantic representations, called *signatures*, are stored in the case base to be matched by queries.

```

Procedure InitializeCaseBase()
  For each decisive point example  $(X_i, f(X_i))$ , add the example to CaseBase
  Get the match criteria,  $Criteria \leftarrow GSA(CaseBase)$ 
  For each case  $X_i$ 
     $\vec{M}_i \leftarrow BuildMatchVector(Criteria, X_i)$ 
  Train NeuralNet( $\vec{M}_i$ ) to map  $\vec{M}_i \rightarrow f(X_i)$ 
  Associate  $Signatures_i \leftarrow NeuralNet(\vec{M}_i).HiddenLayer$  with each  $X_i$  in CaseBase
  Return  $Criteria$ 

Procedure BuildMatchVector( $Criteria, Case$ )
  For each  $Criteria_j$ 
    
$$V_j = \begin{cases} 1.0 & \text{if } Criteria_j \text{ is true in } Case \\ 0.0 & \text{otherwise} \end{cases}$$

  return  $\vec{V}$ 

Procedure GenerateDecisivePoints( $Criteria, CaseBase$ )
  For each candidate decisive point  $P_j$ 
    Query PowerLoom rules to determine  $Goodness_j$ 
    If  $Goodness_j$  cannot be determined from rules
       $\vec{V} \leftarrow BuildMatchVector(Criteria, P_j)$ 
       $Query_j \leftarrow NeuralNet(\vec{V}).HiddenLayer$ 
       $Nearest \leftarrow 3 \text{ Euclidean nearest cases when comparing } Query_j \text{ to } Signatures$ 
       $Goodness_j \leftarrow WeightedAverage(f(Nearest_1), f(Nearest_2), f(Nearest_3))$ 
    Sort  $\vec{P}$  by  $Goodness$ 
  Return top 3 points from  $\vec{P}$ 

```

Figure 6: Decisive point algorithm

Given a new scenario, the algorithm evaluates each map feature (e.g., geographic regions, military units, cities, etc.) as a potential decisive point. Given a map feature, the algorithm first queries PowerLoom to determine if any rules match this feature and can infer goodness. If goodness cannot be inferred through a rule, the algorithm invokes the case-based reasoner. The trained neural network computes a signature for the map feature which is subsequently compared to the signatures in the case base using standard

Euclidean distance. The goodness of the map feature is the distance-weighted average of the goodness of the top three closest matching cases. Once all features have been evaluated, the top three points are returned.

The case-based algorithm follows a general *k-nearest neighbor* strategy with several innovations for managing structured cases and existing rule knowledge. The remainder of this section describes these enhancements and how they contributed to the success of the decisive point reasoner.

3.3.1 Generating Match Criteria

To judge similarity between two cases, one must know the dimensions along which cases may vary. Unfortunately, as described in Section 3.2.3.1, such criteria is not obvious when using relational case representations. This section describes an algorithm called GSA (Generalized Structural Assertions) that uses a heuristic approach to generate match criteria from a relational case-base. GSA capitalizes on one simple idea: the most important criteria for judging similarity between any two cases is the set of facts that have been used to describe the cases. In other words, rather than including all predicates that could be used to describe a case, GSA only includes predicates that actually have been asserted.

The GSA algorithm can be broken down into two main phases. In the first phase, GSA collects and generalizes the set of asserted facts describing each case in the case base. GSA traverses the structure of each case depth-first to a given depth and records all links. The algorithm follows that of Emde [1996] and is as follows. Starting with the root instance of the case, generate and record all directly asserted facts. Repeat this process for each new instance found in the new facts until a specified depth limit d is reached. The knowledge structure in Figure 5 represents the asserted facts at depth limit two for a case rooted at EA1.

After collecting the facts, GSA generalizes the facts, creating a *generalized knowledge structure*. Generalization broadens the scope of the assertions to apply to many cases rather than one specific case. GSA generalizes by substituting variables for instances.

Specifically, variables are substituted for all instances linked to the root instances with less than d relations. In other words, all non-leaf instances are variablized.

Figure 7 shows the generalization of the Figure 5 knowledge structure. By variablizing the assertions, other cases can be matched into the structure simply by generating bindings for the variables. For example, a case might be similar to the EA1 decisive point, because it is near a river and is part of a protect task. Without generalization, a matching case would have to be near River5 and be part of Mission1.

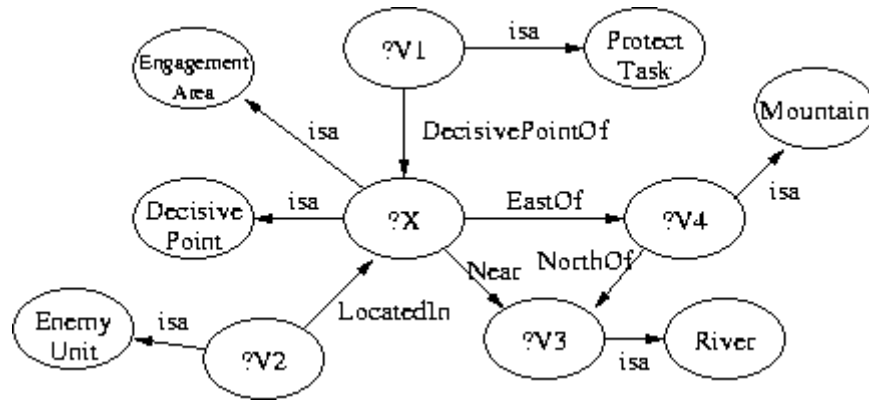


Figure 7: A generalized knowledge structure. Variables are substituted for all non-leaf instances.

The second stage of GSA concerns folding in generalized knowledge structures from multiple cases. A naive combination strategy would simply attach each structure under a common root node. This strategy, however, ignores overlap between knowledge structures and creates an unnecessarily complicated unification. A better strategy when folding in a new knowledge structure is to only add structure that is not present in the unified structure. Unfortunately, finding the largest common overlap among knowledge structures is an instance of the largest common subgraph problem, which is known to be NP-Hard. Therefore, GSA uses a heuristic algorithm to find the common knowledge structure and does not guarantee the simplest unification.

Figure 8 illustrates the GSA structure combination process. GSA starts unifying at each root node ($?X$ in Figure 7) and moves down the graph attempting to align variables. The GSA folding algorithm is as follows. First, sort the generalized assertions based on the

distance from the root node. The distance of an assertion from the root is the smallest number of relational links that connect a single argument in the assertion to the root variable. For example, in Figure 7 the assertion *NorthOf*(*?V4*,*?V3*) has a distance of 1, since *?V3* can be linked to the root, *?X*, using a single relation *Near*. Sorting ensures that GSA unifies all variables close to the root before moving to the assertions deeper in the graph.

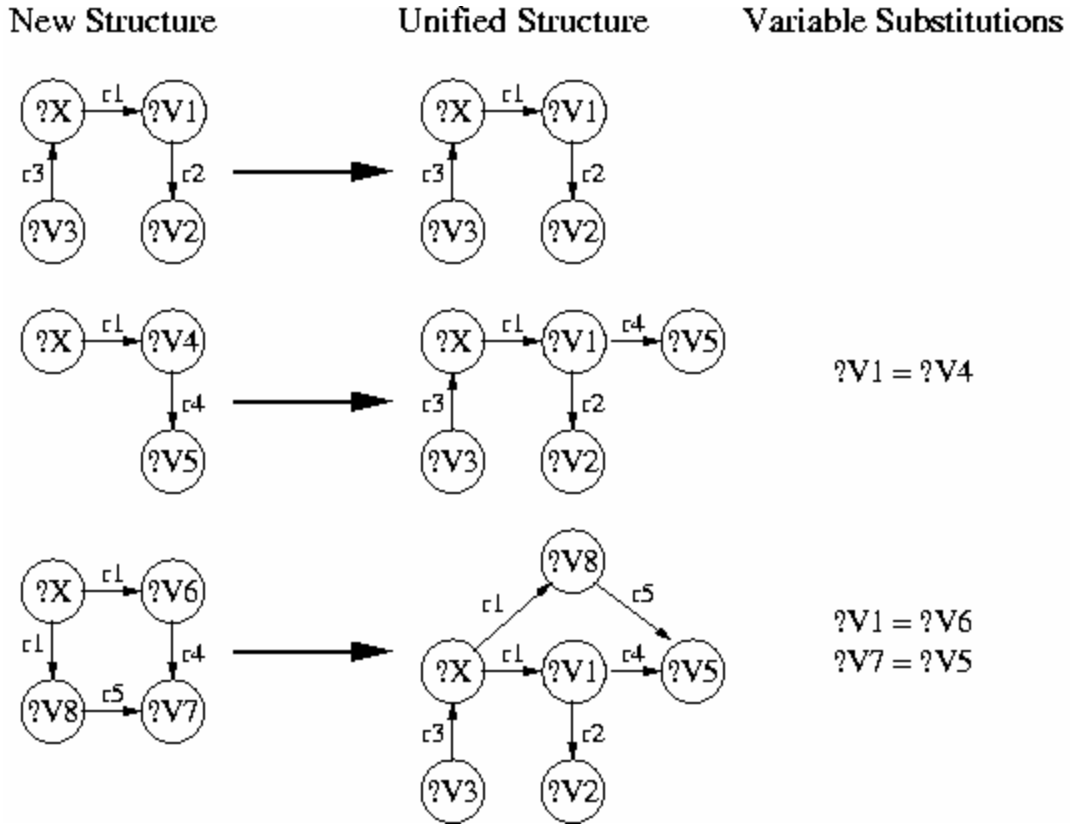


Figure 8: An example of structure folding in GSA. The figure illustrates folding three different knowledge structures into one unified structure. The first structure is simply copied to the empty unified structure. In the second example, variable *?V4* is unified with variable *?V1*, since they both share the relation *r1* from *?X*.

Second, for each assertion in the sorted knowledge structure attempt to find a matching assertion among the unmatched assertions in the unified knowledge structure. Two assertions match if they contain the same predicate and there is no conflict between their

arguments. Argument conflicts occur when a common variable between the two assertions appears in different argument positions. The result of the assertion match is a set of variable substitutions necessary to complete the match. For each match, GSA deletes the matching clause from the sorted knowledge structure and propagates the variable substitutions through the remaining clauses. The final folding step adds all unmatched assertions in the sorted knowledge structure to the unified knowledge structure. The unmatched assertions represent new knowledge that is not currently represented in the unified knowledge structure.

The result of GSA is a knowledge structure that reflects the combined descriptions of the cases. The structure can be thought of as the set of relevant criteria to describe a case. In the next section, we will exploit this notion by using this criteria as a foundation for judging similarity between any two cases.

3.3.2 Building Semantic Signatures

A key problem with relational case knowledge comes at query time, when a query must be compared to all cases. The problem is that structured cases are difficult and time consuming to compare. Even with the criteria generated by GSA, structured case matching entails finding the largest common overlap between two graphs, which as mentioned before is an NP-Hard problem. With a large, complex case base, a structure matching algorithm requires significant computational overhead and is likely impractical.

One way to reduce the computational overhead is to use a preprocessing step to filter out cases that are unlikely to be relevant to the query and thereby reduce the number of calls to the structural matcher. Gentner and Forbus [1991] describe a system that uses an efficient literal similarity test to filter out unrelated cases. The disadvantage of this approach is that it introduces a weaker comparison algorithm that may miss important similarities. Thus, some cases that the structural matcher finds similar may not be returned because the weaker algorithm weeded them out.

An alternative strategy is to translate the structural cases into representations that can be easily and efficiently compared. One representation conducive to efficient comparisons is a fixed-length vector of floating point numbers. Floating point vectors represent points

in a multidimensional space and can be compared using simple Euclidean distance. The challenge is thus to translate a case represented by logical assertions into a fixed-length continuous vector, while preserving as much of the original semantics as possible.

The remainder of this section describes an algorithm for performing such a transformation using PowerLoom's deductive reasoners and a neural network. The approach generates *signatures* for each case which are compact, semantically-rich floating point representations. Figure 9 illustrates the major steps.

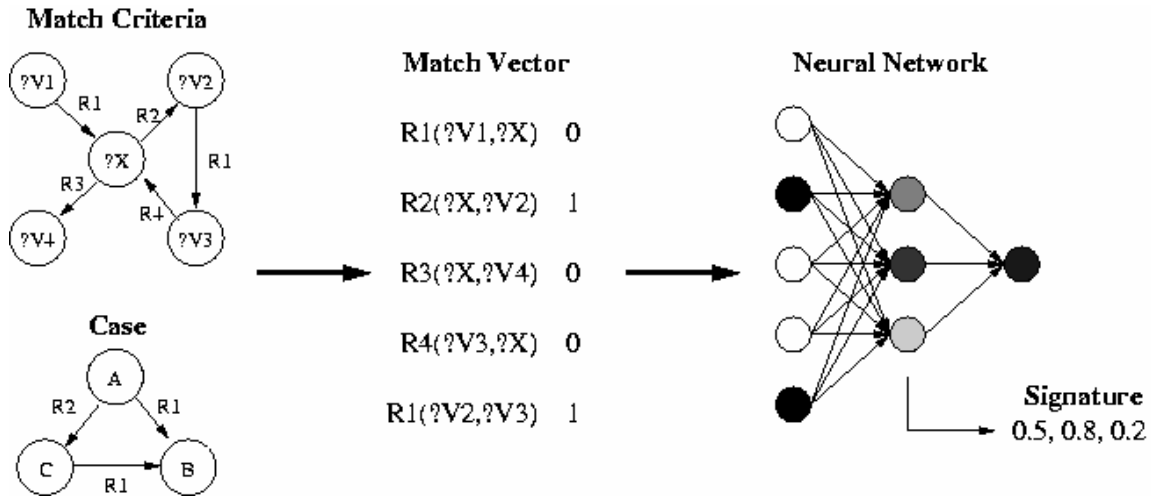


Figure 9: Transformation of a structural case into a floating point signature.

3.3.2.1 Creating Match Vectors

In the first stage of signature algorithm, PowerLoom performs a structural match between each case and the match criteria. PowerLoom matches each case by binding variables in the match criteria to instances in the case. Since it is unlikely that any case will match the entire match criteria, PowerLoom uses a greedy partial-match strategy where it binds variables such that the greatest number of bindings can be made. In other words, PowerLoom finds the greatest overlap between the case and the match criteria.

The result of the PowerLoom match for a given case is a set of clauses from the match criteria that are satisfied and a set that are not. For example in Figure 9, only two of five clauses, $R2(?X, ?V2)$ and $R1(?V2, ?V3)$, are satisfied by the case. By associating a score of

1 for each satisfied clause and a score of 0 for each unsatisfied clause, we obtain a pattern or footprint for the case within the match criteria. These patterns are essentially fixed-length binary vectors that represent which clauses in the criteria are satisfied by the given case. We will subsequently refer to these patterns as *match vectors*.

3.3.2.2 Neural Network Feature Weighting

The match vectors provide a fixed-length representation that captures the semantics of each case. What is missing, however, is a measure of importance for each dimension. Clearly, when comparing two cases, some criteria should be weighed more than others. For example, the type of mission is likely to be crucial in determining decisive points and should be weighed more in comparisons than superficial properties such as the names of the commanders of a unit. Numerous methods have been proposed for feature weighting in case-based reasoning [Wettschereck, Aha & Mori, 1997]. Most approaches attach a weight to each feature and compute the match score by summing the weights of the matched features. Traditionally, effective weights are found through some sort of hill-climbing search over the weight space. We adopt a similar strategy, but use a neural network rather than linear combination to compute the match score.

The goal of the neural network learning module is twofold. First, it learns which features are most relevant for evaluating decisive points and consequently how to weigh them when computing a match. Second, it computes a compact, semantically-rich representation that can be easily matched to other cases. The neural network is a standard 3-layer feedforward neural network with sigmoid units in the hidden and output layer. The input to the network is the match vector for a given case. The output of the network is the expert's evaluation of that case (a continuous value between 0.0 and 1.0), and the network is trained using the standard backpropagation algorithm to return the correct evaluation score for every case in the case base.

Once the network has been trained, it has learned to classify all of the decisive points in the case-base based on the input patterns within the match criteria. Thus, it has automatically learned how to weigh the different features in the input when classifying decisive points. Since the neural network uses a hidden layer of units, it has also learned to translate the binary input patterns into a lower-dimensional continuous space

represented by the hidden units. The hidden unit activations capture the features of the input patterns that are important for classifying decisive points. Our strategy is to use these activations as a semantic representation or *semantic signature* for each case.

There are several advantages of the neural network semantic signature strategy. First, it generates a more compact representation for each case, which for large case bases significantly reduces the match time. In this problem, the neural networks reduced 500 dimensional match vectors to 50 dimensional vectors. Second, neural networks can weigh mismatch evidence as well as match evidence. Current linear combination strategies only propagate positive match evidence from each feature, which in the decisive point problem is not always valid. For example, two bridge examples may match exactly, except that the military unit on each bridge belongs to different sides. This difference should completely change the similarity measure since in one case the bridge is controlled by friendly forces, and in another case it is controlled by the enemy. It would be difficult to craft a set of linear feature weights to make such a large distinction from a single mismatched feature. The neural network strategy can, however, separate these examples based on the one mismatch by mapping them into different areas of the hidden unit space.

To summarize, we have developed a methodology for matching structural cases that builds flat semantic signatures that can be easily matched using Euclidean distance. One key advantage is match time. While our strategy still requires structure matching, it significantly reduces the frequency. Ignoring training for feature weighting, a structural case-based reasoner requires $O(CQ)$ structural matches, where C is the size of the case base and Q is the number of queries. Our semantic signature approach requires only $O(C + Q)$ structural matches. Since queries normally become cases, we've reduced the number of structural matches from polynomial to linear in the number of queries.

3.3.3 Incorporating Rule Knowledge

The decisive point algorithm uses existing rule knowledge in two important ways. First, before invoking the case-based reasoner on a given map feature, it explicitly checks if any existing rule knowledge can infer goodness. If a rule exists that covers the feature, it foregoes the case-base strategy and uses the rule to assess goodness. An example of this

type of knowledge is the rule: *linear features are bad decisive points*. All rivers and roads match this rule and are given poor evaluation scores without querying the case base.

A second way that the decisive point algorithm uses rule knowledge is in structural case matching. Recall from the previous section that PowerLoom's partial matcher is used to match each case into the match criteria. Structural matching entails finding bindings for variables in the match criteria such that as many clauses are satisfied as possible. When satisfying a clause, PowerLoom uses any available inference rules to generate a deductive proof. For example, suppose a clause in the match criteria specifies *East(?X,?V1)*, but there is no corresponding assertion in the case. Suppose that the case does have the assertion *West(River4,Bridge1)*. Given the general rule $East(X,Y) \Rightarrow West(Y,X)$, PowerLoom can infer *East(Bridge1,River4)* and satisfy the above clause.

Utilizing rule knowledge in case matching compensates for a lack of cases. In the previous example, the rule creates a second implicit case from an explicit case. In other words, there was no case where *East(Bridge1,River4)* was explicitly asserted, but the rules allowed PowerLoom to infer one. Experiments in the evaluation section confirm the importance of this kind of knowledge when cases are limited.

3.4 Evaluation

Identifying good decisive points provides an interesting challenge and knowledge-rich case-based reasoning appears to be a promising solution. To test this hypothesis, we participated in an official HPKB evaluation conducted by the Alphatech Corporation.

Alphatech provided expert analysis of 50 decisive point cases over 3 different scenarios and 9 different missions. The cases were evenly split between positive and negative examples. Each case was modeled using terms from ontologies produced within HPKB, which includes a portion of the Cyc knowledge base. To complete each case, we used background knowledge about each scenario including the geospatial information of all map features and specific knowledge of each military unit. It is worth mentioning that all background knowledge was modeled by others in the HPKB community and is thus independent of our specific reasoner.

For the evaluation, Alphatech provided five different problems. Table 1 summarizes our performance as determined by Alphatech's military experts. Since our algorithm returns the top three decisive points for any scenario, Alphatech provided two measures of performance. The first measures the quality of the best decisive point that we returned and the second measures the average of the three. The graph shows that in every problem, we returned a very good decisive point. The lowest "best" score was 80% in problem 4. This result is very encouraging and shows that the reasoner recognizes the best decisive points.

Problem	1	2	3	4	5
Best	100	100	100	80	100
Average	100	100	80	40	95

Table 1: Results from the HPKB evaluation. Scores range from 0 to 100.

Unfortunately, the average scores shows that the system was not as discriminating with poor decisive points. For example in problem 4, while the system did return the optimal decisive point, it also returned two decisive points that were judged very poor. From the evaluation numbers and expert opinions, it is clear that the system is overgeneralizing. It does not miss the best points, but it does not always reject the bad ones. An obvious solution to this problem is to provide more negative examples.

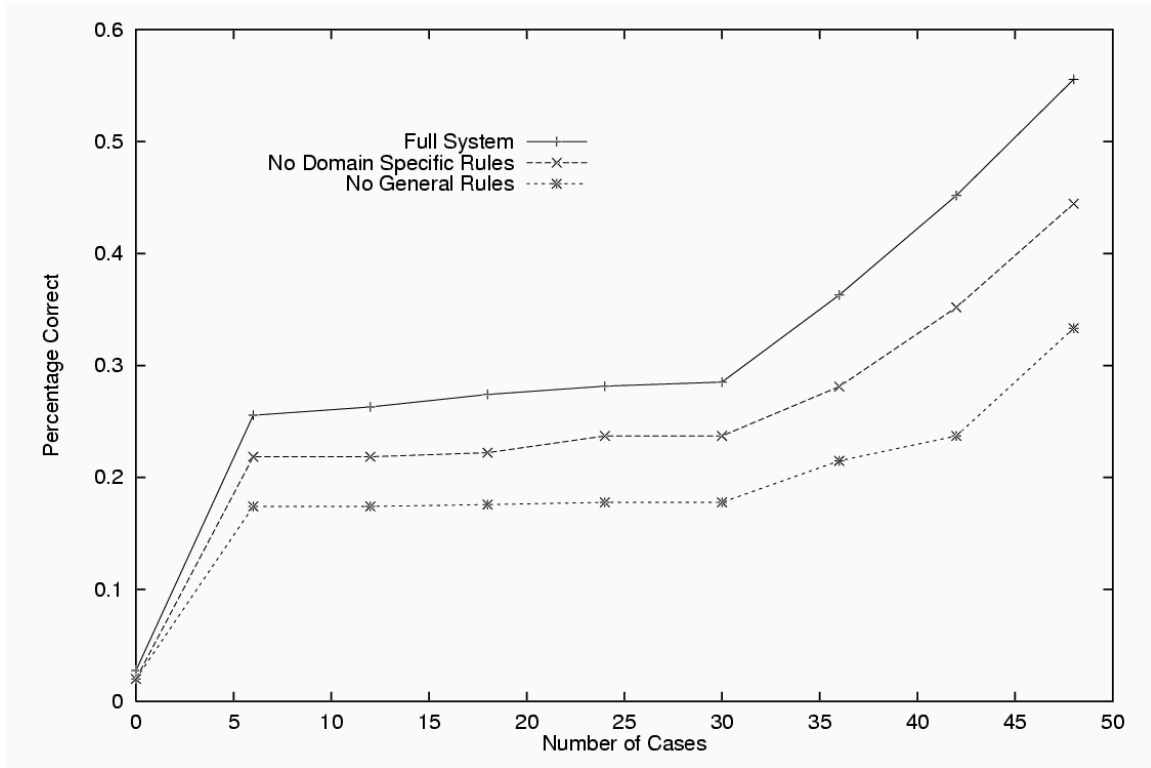


Figure 10: Learning curves for different variations of the case-based reasoner.

To complement the Alphatech evaluation, we ran additional experiments in-house to both judge performance and measure the utility of the rule-based knowledge. Figure 10 shows the results of several 10-fold cross validation experiments over the case base with and without rule knowledge. The learning curves plot the rate at which performance increases with the size of the case base. The top curve represents the full system with all available rule knowledge. The middle curve plots performance without knowledge specific to decisive points such as the fact that rivers are not good decisive points. The bottom curve plots system performance without domain-general rules such as general geospatial relationships.

The shape of the curves are somewhat surprising and do not reflect typical machine learning curves. The most striking feature is that the learning rate actually increases with experience. There is virtually no difference in performance with case base sizes of 5 to

30 examples, but after 30 the performance grows approximately linearly with the size of the case base. Unfortunately, we have not been able to come up with a concrete explanation for the shape of the curves. One possible hypothesis is that they are an artifact of the cases. It is unclear how representative our case distribution is over the actual problem distribution.

The results do show a significant advantage to case-based reasoning with rule-based knowledge. At every level of case knowledge, performance improves with rules, which supports our hypothesis that the rules compensate for a lack of cases. The knowledge-poor approaches need more cases to achieve the same level of performance as the knowledge-rich approach. Interestingly, there was a greater drop off without the general rules than without the domain specific rules. Since general rules apply to more situations, their impact is felt more often and thus when they are removed the system suffers a greater performance hit.

The overall assessment by the military experts is that our approach is the most promising solution to the decisive point problem to date. The system is currently not strong enough to serve in real planning efforts, but the experts agree that the currently limiting factor is the lack of cases, not the technology. Given more cases (especially negative examples) and additional rule-knowledge, the knowledge-rich case-based reasoner could provide a valuable military planning tool.

3.5 Conclusions

Determining decisive points is a challenging military problem, where the lack of expert rules precludes a complete expert system or rule-based solution. Case-based reasoning offers a promising solution since knowledge is acquired from examples rather than rules. However, traditional methods are inadequate because they cannot match structured cases and do not incorporate existing rule knowledge. We addressed each of these challenges in a new case-based reasoner within the PowerLoom knowledge representation system that we characterize as knowledge-rich case-based reasoning. The PowerLoom case-based reasoner combines nearest neighbor, graph search, neural networks, and natural deduction to learn effective match criteria, perform structural matches, and leverage off

of existing rule-based knowledge. Experiments in the HPKB program show that our approach is more effective than any other decisive point solution to date.

4 Java-Based Graphical Knowledge Editor

4.1 Overview

Below we describe the PowerLoom knowledge editor (or GUI), a Java-based graphical client for the PowerLoom Knowledge Representation and Reasoning System. We first describe the architecture of the PowerLoom GUI, and discuss design issues and tradeoffs. Next, we present an overview of the GUI, highlighting high-level functionality. We then present a comprehensive list of the features and capabilities that are currently present in the GUI. Finally, we conclude with some possible future directions for the GUI.

4.2 Architecture

4.2.1 Client/Server Architecture

In the traditional client/server (a.k.a. 2-tier) model, a “smart” client application communicates with a “dumb” datastore such as an RDBMS or file system. More recently, the 3-tier (or “n-tier”) model has become popular, in which a “dumb” client such as an HTML browser communicates with a “smart” middle tier which contains the application's business and presentation logic, and which in turn communicates with a back-end datastore. Likewise, in the 1-tier model, a client and server share the same process space. An example of a one-tier application is a word processing application which stores data in a flat file.

The architecture for the PowerLoom GUI most closely resembles the client/server model. The GUI component is a Swing-based Java client which communicates with a remote version of the PowerLoom KRRS. Since the GUI does not contain a great deal of business logic (e.g., it does not know how to do inferencing), it does not directly map onto the traditional notion of a smart client. Similarly, since PowerLoom is much “smarter” than a typical DBMS, it does not cleanly map onto a traditional backend server. However, since the GUI contains the presentation logic, it is more similar to a 2-tier model than a 3-tier model in which the presentation logic resides on the middle tier.

The GUI has been designed in such a way that it can be “baked-in” with a Java version of PowerLoom. In this mode, the client runs in the same process as the server, and bypasses

the SOAP communication layer. Although we have not yet experimented with this mode, it is likely that its use could result in significant performance benefits for standalone applications.

Communication between the GUI and PowerLoom is done via the XML-based SOAP protocol. In order to effect communication via SOAP, a Web service layer was built on top of PowerLoom. This layer provides support for marshaling and unmarshaling of PowerLoom objects to/from XML, and also provides a PowerLoom API that is accessible as a web service. The Java client uses JAXM and the Castor framework (see <http://www.castor.org>) to support SOAP communication.

The GUI can be launched either from a shell prompt or over the web using Java Web Start (JWS) technology. With JWS, users are required to do a one-time download and install of a JWS client application, after which they can launch any fully-functional Java application over the web. This technology overcomes many of the problems associated with Java Applets, including their restrictive set of capabilities and browser compatibility issues.

4.2.2 GUI Design Goals

At the outset of our design effort, our biggest decision was whether to implement the PowerLoom interface as a Swing GUI or as an HTML-based web application. Ultimately, we decided that a Swing GUI better suited our needs than a Web application. In arriving at this decision, we considered the following requirements:

4.2.2.1 Visibility

Knowledge Bases are complex and loosely structured entities. It is often desirable to simultaneously maintain multiple views of a KB, and to simultaneously perform multiple complementary tasks such as browsing, editing, querying, and searching a KB. We feel that Swing-based applications are superior to HTML-based applications when it comes to displaying large quantities of complex information. With Swing's MDI (Multiple Document Interface) mode, many internal frames can be open at the same time within a single “desktop” frame. Swing also offers a rich set of components and UI mechanisms which facilitate efficient use of screen real estate. These include the ability to resize,

close, and iconify windows, layout algorithms which are smart about redisplaying internal components, and components such as collapsible trees, scrollable subpanes, movable dividers, tabs, etc. In contrast, HTML has no similar MDI capability, and simulating widgets such as trees can be a cumbersome programming chore.

We designed the GUI to take advantage of Swing's presentation strengths. We used the MDI mode, so browsers, editors, etc. can coexist on the same desktop. Additionally, multiple knowledge browsers can be open at the same time to present different views of a KB. The Knowledge Browser itself consists of multiple collapsible and resizable subpanes, which in turn are composed of scrollable lists and trees. This allows a “birds-eye” view of a Powerloom KB, in which many modules, concepts, relations, instances, propositions, and rules can be displayed at the same time.

4.2.2.2 Navigability

When exploring a KB, it is imperative that a user interface allows easy navigation between related objects. HTML-based applications are excellent for applications that require navigation capabilities, since the primary function of a hyperlink is to navigate to a new HTML page. Implementing navigation in Swing requires a bit more coding effort.

The PowerLoom GUI has extensive navigation capabilities, which are as good or better than browser based applications. For example, a user may click on a query result to instantly update the Knowledge Browser to display the selected object. Also, a user can right-click on a relation or argument in a proposition, and navigate to the clicked-on object.

4.2.2.3 Responsiveness

For the best possible user experience, a user interface should be highly responsive to a user's input gestures. This is true in two respects: 1) After initiating a gesture such as a mouse click or keypress, there should be a minimal delay before the application performs the intended action, and 2) “Power Users” should be able to perform complex tasks with a minimum number of mouse clicks, key presses, etc. In general, Swing is more responsive than HTML browsers on both counts: 1) Since a significant amount of cached state is maintained in a Swing client, there is less need to do network round-trips to

retrieve information, and 2) Swing has many mechanisms to enable efficient control of an application, including menu accelerators and programmable keymaps that are associated with components.

The PowerLoom GUI attempts to minimize network round-trips by caching large amounts of data. For example, when a user points the Knowledge Browser to a module, a large chunk of the module is retrieved from the server and cached in the client. Hence, when the user expands a tree in the browser, the GUI will not need to retrieve more data from the server. Also, the GUI takes full advantage of Swing's ability to control the application via keyboard input. For example, to create a new instance named `newName`, a user needs to simply type the key sequence: `CTRL-I newName [RETURN]`.

4.2.2.4 Context Sensitivity

For any given object that is displayed in a user interface, there is a set of actions that can be performed on that object. Additionally, the actions that can be performed on an object depend on where the object is displayed. In a browser-based application, there is only a single action that can be performed on a displayed object, i.e., the action that is executed when the object's hyperlink is clicked. In contrast, Swing enables the use of context-sensitive popup menus. When a user right-clicks on an object, a list of appropriate actions will be presented in a menu.

Context-sensitive menus are ubiquitous in the PowerLoom GUI. For example, when a user right-clicks on a concept in the Knowledge Browser, they are presented with the following list of possible actions:

- Add a new concept
- Edit the concept
- Edit the concept's extension
- Instantiate the concept
- Cut the concept
- Copy the concept
- Paste a concept

- Delete the concept.

4.2.2.5 Editability

Applications that support text editing often need capabilities above and beyond the baseline capabilities that all text widgets support: cut, copy, and paste. In particular, applications that allow editing of text with a regular structure such as source code or Lisp expressions may take advantage of special key bindings which augment basic navigation and editing capabilities. HTML browsers offer no means of enhancing a browser's basic text widget. Swing, on the other hand has very powerful text components which allow keys to be bound to arbitrary actions.

The PowerLoom GUI makes use of Swing's powerful text components by implementing a full set of Emacs-style keybindings. These keybindings allow a user to perform such operations as navigating up and down a subexpression hierarchy, selecting entire subexpressions, and completing incomplete symbols. In addition, matching parenthesis are automatically highlighted in the GUI's text components.

4.2.2.6 Extensibility

While it is not easy to claim that Swing applications are inherently more extensible than Web applications, Swing's MDI architecture and pull-down menu framework allows new features to be added with little disruption to the rest of the application.

With the aid of a GUI design tool such as Sun's Forté, new internal frames can be easily added to the PowerLoom GUI. We envision that additional tools such as KB Graphers or other KB visualization or analysis tools could be added to the GUI in the future. Also, it is conceivable that the GUI code could be used as a basis for a more specific application. The application would have its own application-specific menus and windows, but would retain the general-purpose browsing, querying, and editing tools for direct manipulation of the knowledge base. It should also be noted that the implementation of PowerLoom's Web service interface should facilitate rapid integration of applications with PowerLoom.

4.3 GUI Overview

The PowerLoom GUI is shown in Figure 11. The main application frame consists of pull-down menus, a toolbar, and a status bar.

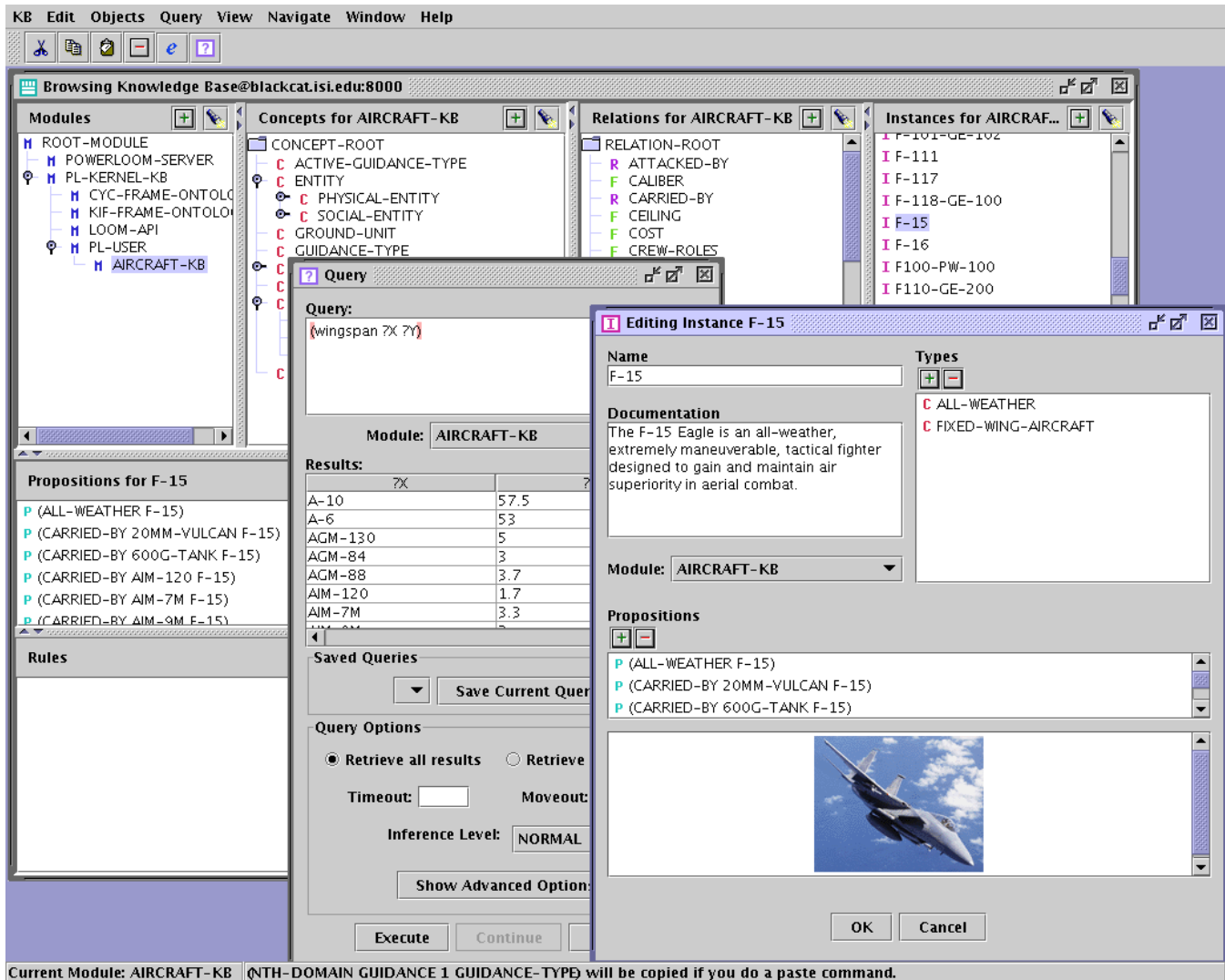


Figure 11: The PowerLoom GUI

The application has the following menus: KB, Edit, Objects, Query, View, and Navigate. These menus are described as follows:

- **KB** - The KB menu contains menu items for connecting to a server, loading, saving, and clearing KBs, and opening browser and console windows.

- **Edit** - The Edit menu contains items for cutting, copying, pasting, and deleting, and also contains an item which opens a preferences dialog.
- **Objects** - The Objects menu contains items for opening editors on various KB objects, including modules, concepts, relations, instances, and propositions. This menu also contains an item to edit the currently selected object.
- **Query** - The Query menu contains items for querying the KB, searching the KB, and editing a relation's extension.
- **View** - The View menu contains various items for updating the appearance of the application, including a refresh item to bring the GUI up-to-date with the state of the KB on the server, and menu items for showing/hiding the application's toolbar and status bar. This menu also contains items for changing the application's font – the demo theme changes all fonts to a large bold font suitable for demo presentations.
- **Navigate** - The Navigate menu contains items for navigating backward and forward in a browser's selection history.
- **Window** - The Window menu contains a list of the currently open windows on the desktop. Selecting a window brings the window to the top of the window stack, and if the window is iconified, it is de-iconified.
- **Help** – The help menu contains an item to open an HTML help browser, and an item to open an About box which contains information about the PowerLoom GUI.

Most menu items have accelerator keys that allow an item to be executed by a combination of keystrokes. The detailed operation of each of the menu items will be elaborated further in the GUI Features section.

The toolbar contains several buttons which provide shortcuts to menu items. There are currently toolbar buttons for cutting, copying, pasting, deleting, editing an object, and opening a query dialog. The toolbar may be undocked from its default position by dragging it anywhere on the desktop. It may also be hidden by selecting the View -> Hide Toolbar menu item.

The status bar at the bottom of the application contains information on the current status of the application. The status bar is divided into two sections. The leftmost section

displays the last module that was selected by a user. The application keeps track of the current module in order to provide continuity between operations. For example, if a user opens a browser and browses the AIRCRAFT-KB, and then opens a query dialog, it makes sense for the query dialog to use the AIRCRAFT-KB module instead of some other module.

The rightmost section of the status bar contains messages that pertain to the current state of the application. For example, if a user selects a concept and then clicks the cut toolbar button, a message will appear in the rightmost status bar prompting the user to select another concept and perform a paste action. The status bar may be hidden by selecting the View -> Hide Status Bar menu item.

Figure 11 shows a few internal frames that are open. The function of each frame is identified in the frame's title bar, and each type of frame has a unique icon in its upper left-hand corner. In this example, the three open frames are used to browse the KB, query the KB, and edit an instance, respectively.

A user typically follows a workflow cycle similar to the following sequence:

1. The user launches the GUI by clicking on a hyperlink, executing a shell command, or clicking on a desktop icon.
2. The GUI is loaded on the user's machine. If the GUI was launched via JWS, JWS may need to download the entire application or updates to the application before execution begins.
3. The GUI reads a preferences file stored in a default location on the user's local machine. If this is the first time the application is being executed, a default preferences file is used. The preferences file includes among other things the PowerLoom server that was last accessed.
4. If the preferences file contains the last-accessed server, it attempts to connect to the server and query the server for a description of the server's capabilities. If connection is successful, a browser window will open displaying the modules that are currently loaded in the server instance.

5. The user selects any KB files (s)he wishes to load, and instructs the server to load the file.
6. The user performs some browsing, querying, and editing of the loaded KB.
7. If any changes were made, the user saves the KB.
8. The user repeats steps 5-7 as needed, and then exits the application.

4.4 GUI Features

This section provides a detailed description of the features that are available in the GUI application. We describe general application-wide functionality as well as the functionality of specific components.

4.4.1 Connect to Server

The first time the GUI is started, it will not attempt to connect to any server. To establish a server connection, the user must select the KB -> Connect to Server menu item. This will open a dialog prompting for a host name and port. After the user enters this information, a connection will be attempted. If the connection is successful, the server information will be stored in the preferences file and used next time the application starts up.

4.4.2 Edit Preferences

A preferences dialog can be opened by selecting the Edit -> Edit Preferences menu item. Currently, the only preference that a user can edit is whether or not open a browser when the application is started. The dialog contains a checkbox asking whether or not the preferences should be saved. If the checkbox is not checked, the preferences will remain in effect for the duration of the current session, but will not be in effect when the application is restarted.

4.4.3 KB Load/Save

In its standard configuration, PowerLoom stores knowledge bases via flat files. The GUI has two options for loading and saving KB files. The first option is to load/save files using the local file system, i.e., the file system that is immediately accessible to the user.

This option only works if the PowerLoom server has access to the same file system as the user. For example, this might be true if the server was executing on the same LAN as the user's machine. The second option is to load/save files using the file system that is visible to the PowerLoom server. This option would be used in situations where the user is executing the client on a machine that is not immediately accessible to the server, e.g., over the Internet. Since there are significant security risks with this option, this option is disabled by default. It can be enabled by setting a flag on the server indicating that remote file browsing is permissible. In order for this feature to be usable in practice, we would have to significantly enhance the security capabilities of the PowerLoom server and the GUI.

4.4.4 Browsing

4.4.4.1 Overview

The knowledge browser window, shown in Figure 12, can be opened by selecting the KB -> Browse menu item or typing CTRL-B. The browser provides a visual overview of all knowledge in the KB, and is capable of launching specialized tools such as editors, search dialogs, etc.

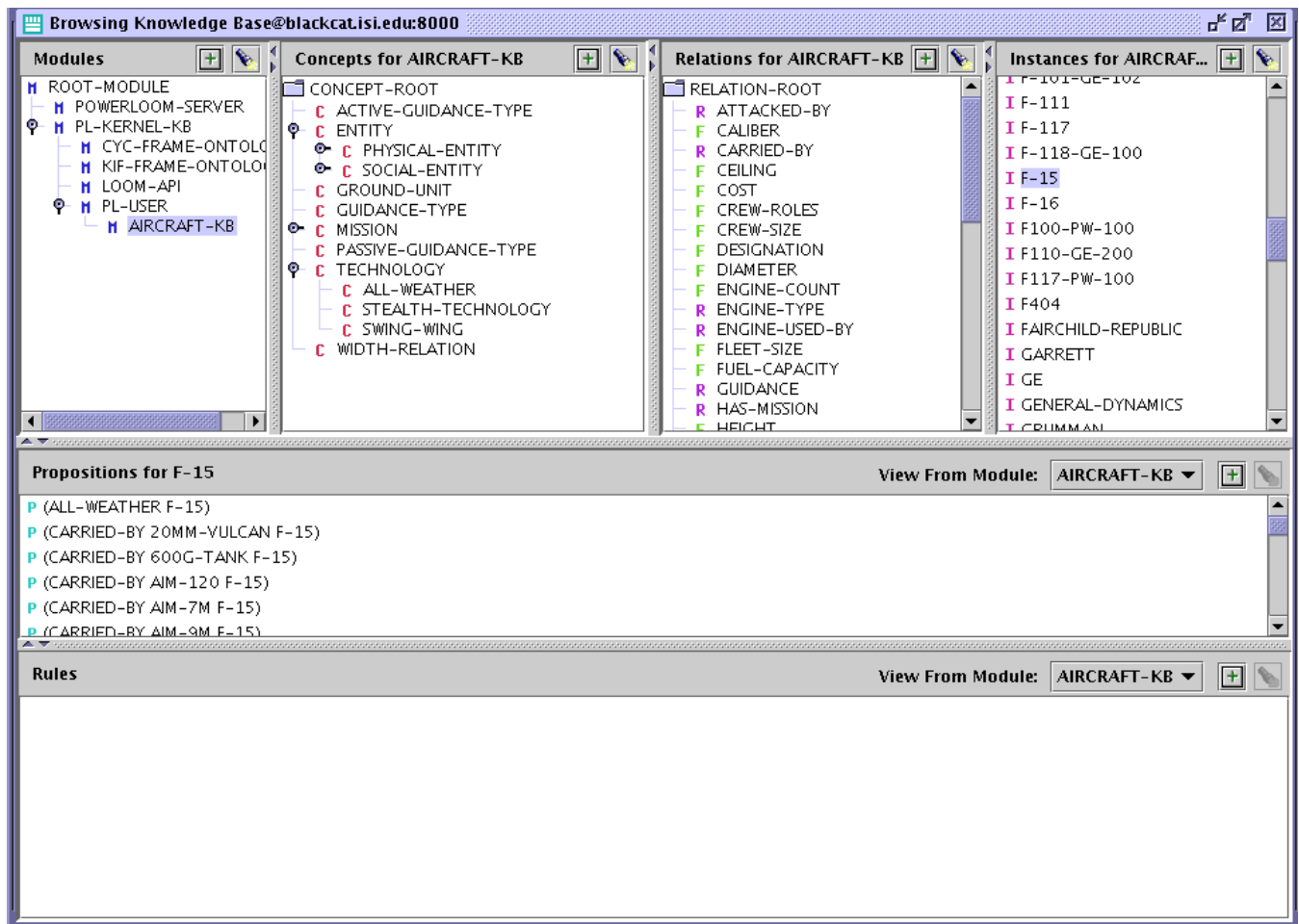


Figure 12: Knowledge Browser

The browser consists of several subpanes which we refer to as navigation panes. Each navigation pane consists of a title pane, a toolbar, and a content pane. The title pane contains a title indicating what is displayed in the content pane. The toolbar consists of zero or more buttons which perform actions relevant to the navigation pane. Currently, two toolbar buttons are present: Add and Search. Add adds an object associated with the type of navigation pane, and Search searches for objects associated with the type of navigation pane. The content pane contains the actual knowledge to be displayed, such as a list of instances or propositions.

There is one navigation pane for each type of KB object: Modules, Concepts, Relations, Instances, Rules, and Propositions. Each internal pane is resizable by dragging the movable divider between the panes. Panes may be hidden completely by clicking the

“collapse” arrow on the adjacent divider. Clicking the “expand” arrow will unhide the pane.

4.4.4.2 Viewing

Navigation panes employ several visual cues to enhance the identifiability of object attributes. Object types are indicated by an icon to the left of the object's name. For example, modules are represented by a blue M, concepts by a red C, etc. The status of propositions is also indicated visually. An Italicized proposition indicates that the proposition was derived instead of asserted. Grey propositions indicate that their truth value is a default value instead of a strict value.

The main method for filling the contents of a navigation pane is to select some object in a navigation pane that is to the left or above it. This is discussed in more detail in the section below. However, in some cases, it is possible to modify the contents of a navigation pane without performing a selection. For example, in the instance navigation pane, it is possible to show derived or inherited instances by right-clicking on the instance list and selecting an appropriate menu item. Similarly, the relation navigation pane can toggle between direct or inherited relations. Propositions and rules are by default displayed according to the module that is currently selected. However, the contents of the proposition or rule navigation pane can be updated by selecting a more specific module in the View From Module combobox contained in the navigation pane's title bar.

4.4.4.3 Selection

When the browser is initially opened, a tree of modules is displayed in the module navigation pane, and all other navigation panes are empty. When a module is selected, the remaining subpanes are populated with knowledge that is contained in that module. Similarly, selecting a concept in the concept navigation pane populates the relation, proposition, and instance panes with knowledge that is relevant to the selected concept. In general, selecting an object in a given navigation pane may affect the contents of navigation panes to the right and/or below it. More specifically, the rules for object selection are as follows:

- Selecting a module populates the concept, relation, and instance subpanes with knowledge contained in the module.
- Selecting a concept populates the relation subpane with relations that use the concept as a domain type, and populates the instance subpane with the concept's extension. The proposition and rule subpanes are populated with propositions and rules associated with the concept.
- Selecting a relation populates the proposition and rule subpanes with propositions and rules associated with the relation.
- Selecting an instance with no selected relation populates the proposition subpane with propositions that refer to the selected instance.
- Selecting an instance and a relation populates the proposition subpane with propositions that contain the relation as a predicate, and the instance as an argument.
- De-selecting an object will update the state of the browser appropriately. For example, after selecting a module and a concept, deselecting the concept will refresh the concept, relation, instance, proposition and rule subpanes to display the knowledge contained in the selected module.

The title pane in each navigation pane displays a description of the source of the subpane's contents. For example, if relation the relation WINGSPAN was selected, and the instance AGM-130 was selected, the proposition subpane would contain the title "Propositions for WINGSPAN and AGM-130".

Each selection event is recorded in a selection history which can be rolled back and forward. For example, assume user selects the AIRCRAFT-KB module and then selects the GUIDANCE-TYPE concept. If the user then selects the Navigate -> Back menu item, the selection history will be rolled back so that only AIRCRAFT-KB is selected. If the user then selects Navigate -> Forward, the selection history will be rolled forward to its original state so that both AIRCRAFT-KB and GUIDANCE-TYPE are selected.

4.4.4.4 Navigation

Knowledge can be explored by expanding and collapsing nodes in hierarchical navigation panes such as the concept and module navigation panes. If a tree or list is not fully visible, the user may use the scrollbar on the navigation pane's righthand side to scroll through the contents of the pane. Detailed views of objects such as concepts and relations can be obtained by right-clicking the object and selecting the `Edit` menu item. To navigate to the constituent of a proposition, the user can right-click the constituent and then select the `Navigate to...` menu item. For example, right-clicking on the `GUIDANCE` argument in the proposition `(NTH-DOMAIN GUIDANCE 1 GUIDANCE-TYPE)` presents a popup menu which displays (among other items) the item `Navigate to GUIDANCE`. Selecting this menu item will cause the browser to display and select the `GUIDANCE` relation.

Actions external to the browser may also update the browser's contents. For example, clicking on an instance in a list of query results will cause the browser to navigate to the selected instance.

4.4.4.5 Actions

Right-clicking inside the browser will present a menu of actions that is relevant to the subpane that contains the mouse pointer. The list of items will depend on whether the mouse is over a specific item or if it is over the background of the subpane's list or tree. For example, when the mouse is over a specific concept, the menu will contain items for cutting, pasting, instantiating, etc., but when the mouse is over the background of the concept's tree, the only menu item presented will be to add a new concept.

The set of actions for each subpane that is available for each subpane is as follows:

- **Module** - `Add Module`, `Edit Module`, `Load (Local/Remote)`, `Save (Local/Remote)`, `Clear`, `Copy`.
- **Concept** - `Add Concept`, `Edit Concept`, `Edit Extension`, `Instantiate`, `Cut`, `Copy`, `Paste`, `Delete`. If multiple concepts are selected, selecting `Create New Concept` from the background menu will create a concept that contains the selected concepts as parents.

- **Relation** - Add Relation, Edit Relation, Edit Extension, Copy, Delete, Show Inherited/Direct Relations.
- **Instance** - Add Instance, Edit Instance, Copy, Delete, Show Direct/Derived Instances.
- **Propositions** - Add Proposition, Edit Proposition, Copy, Delete, Navigate to Constituent, Edit Constituent.
- **Rules** - Add Rule, Edit Rule, Copy, Delete, Navigate to Constituent, Edit Constituent .

4.4.5 Editing/Viewing

4.4.5.1 Overview

Objects may be edited by right-clicking the object and selecting the Edit item menu item in the popup menu. Alternatively, an object may be selected, and then the Objects -> Edit Object menu item can be selected, or the edit toolbar button can be pressed. Object editors do double-duty as object viewers, since all relevant information is present in the editor.

There are several common user actions that are available in edit dialogs. For example, hitting return while the cursor is positioned in the name field of the editor commits the concept. Most editors contain commit and cancel buttons at the bottom which can be used to either commit or abort edits. Lists of items commonly have a + and – button at the top of the lists, which respectively mean add a new item, and delete the selected item. When the + button is pressed, either a chooser dialog (see the Choosers section) or a specialized editor will be opened. Like the browser, list items can be right-clicked to display a list of possible actions. For example, a superconcept can be clicked in a concept editor to immediately edit the concept's parent.

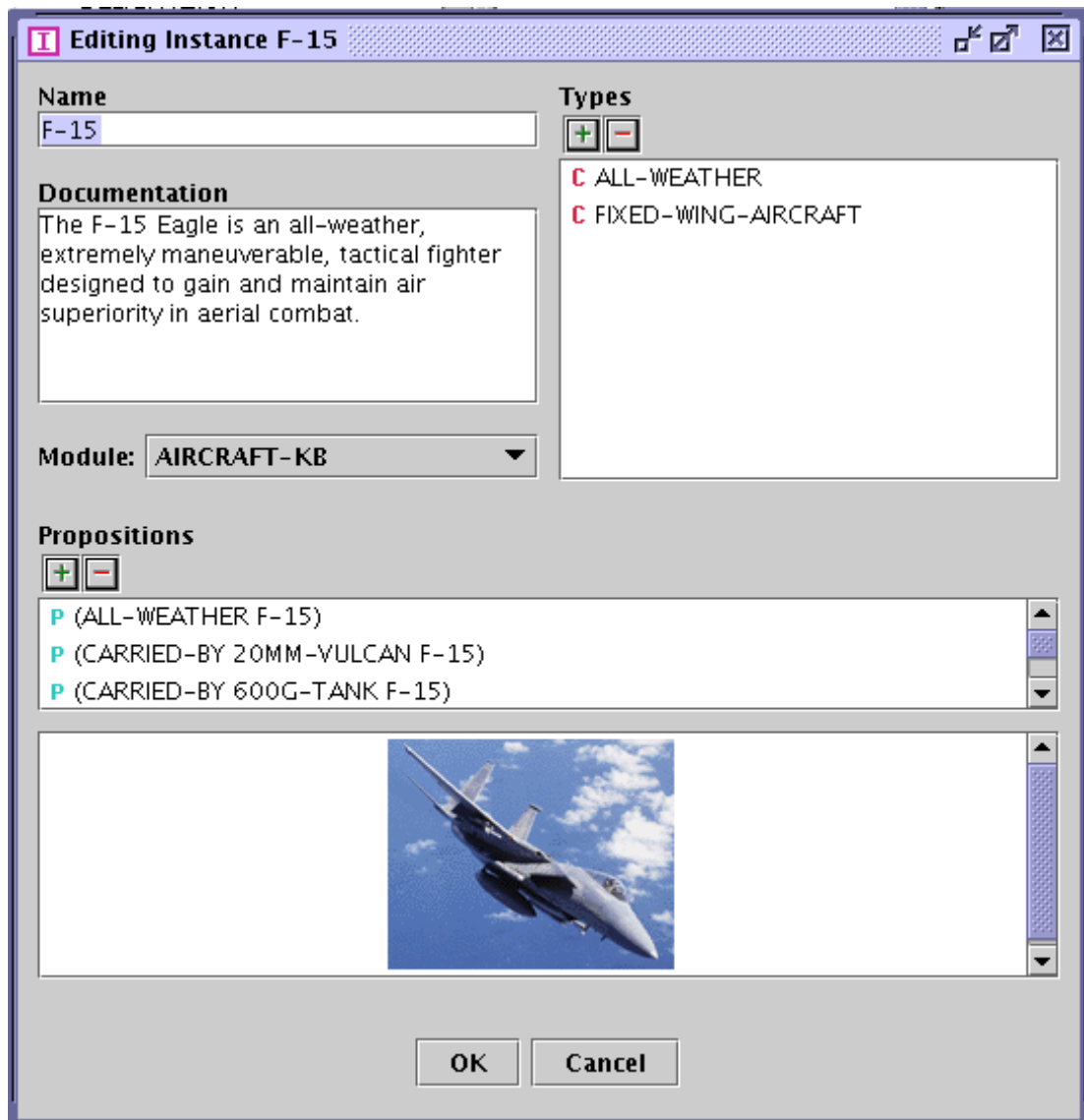


Figure 13: Instance Editor

Each type of object has a specialized editor. For example, an instance editor is shown in Figure 13. There are separate editors for modules, concepts, relations, instances, and propositions/rules, which are described in turn below.

4.4.5.2 Module Editor

The module editor contains a number of fields and components used to enter information relevant for a new or existing module. Examples of values that can be edited are a module's name, a module's documentation, and a module's includes list.

4.4.5.3 Concept Editor

The concept editor allows editing of concept attributes such as a concept's supertypes, its name, its associated propositions, etc. In addition to the inherent attributes of a concept, all relations which have the concept as a domain type are displayed and may be edited. Clicking the + button above the relation list opens a new relation editor, with default values filled in. Similarly, clicking the + button above the proposition list opens a proposition editor.

4.4.5.4 Relation Editor

The relation editor allows the user to input a list of variables and types for the relation's arguments, and allows the user to set various attributes for a relation, such as whether the relation is closed, functional, etc. Like the concept editor, propositions and rules associated with the relation can be edited.

4.4.5.5 Instance Editor

The instance editor allows the user to input an instance's name, documentation, and associated propositions. If a proposition uses the relation image-url, an image will be retrieved from the server and presented in the editor window.

4.4.5.6 Proposition editor

The proposition editor, shown in Figure 14, consists of a text field for entering the proposition, and a set of buttons for performing actions on the proposition. The buttons allow a user to assert, deny, or retract the typed proposition. There are several text-based facilities which support efficient editing of propositions. First, the editor supports many Emacs-style keybindings which facilitate editing of lisp-like expressions, including selecting entire parenthesis-delimited subexpressions, jumping backward and forward over subexpressions, and navigating up and down expression trees.

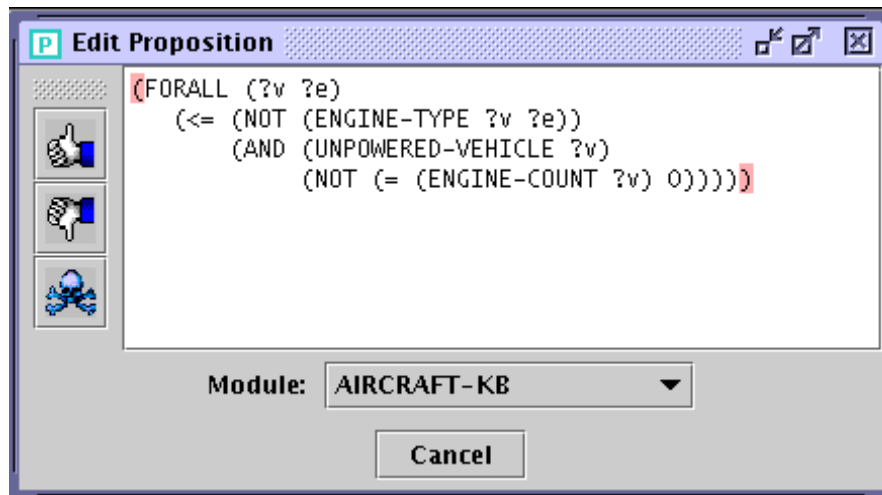


Figure 14: Proposition Editor

In addition to Emacs keybindings, the proposition editor has a matching parenthesis highlighter. When the cursor is placed before a left parenthesis, the matching right parenthesis is highlighted, and when the cursor is placed after a right parenthesis, the matching left parenthesis is highlighted.

The proposition editor also has support for symbol completion. The GUI uses a predictive backtracking parser to analyze partial input of propositions. Based on the analysis, the parser is able to recommend appropriate completions. For example, if the user types (f and then selects the completion action, the parser will recommend a list of completions including the forall symbol and all concepts and relations that begin with the letter f.

4.4.6 Choosers

In a number of situations, an object of a specific type must be selected. For example, when selecting a superconcept in a concept editor, the user should be presented with a list of existing concepts. In these cases, a chooser dialog is presented to the user which displays a filterable list of candidate objects. As the user types a name of the object in the name text field, the list of objects is filtered so that only objects which begin with the typed prefix are displayed. Choosers are available for modules, concept, instances, and relations. A variable chooser allows the user to type a variable name and select a type from a concept from a list.

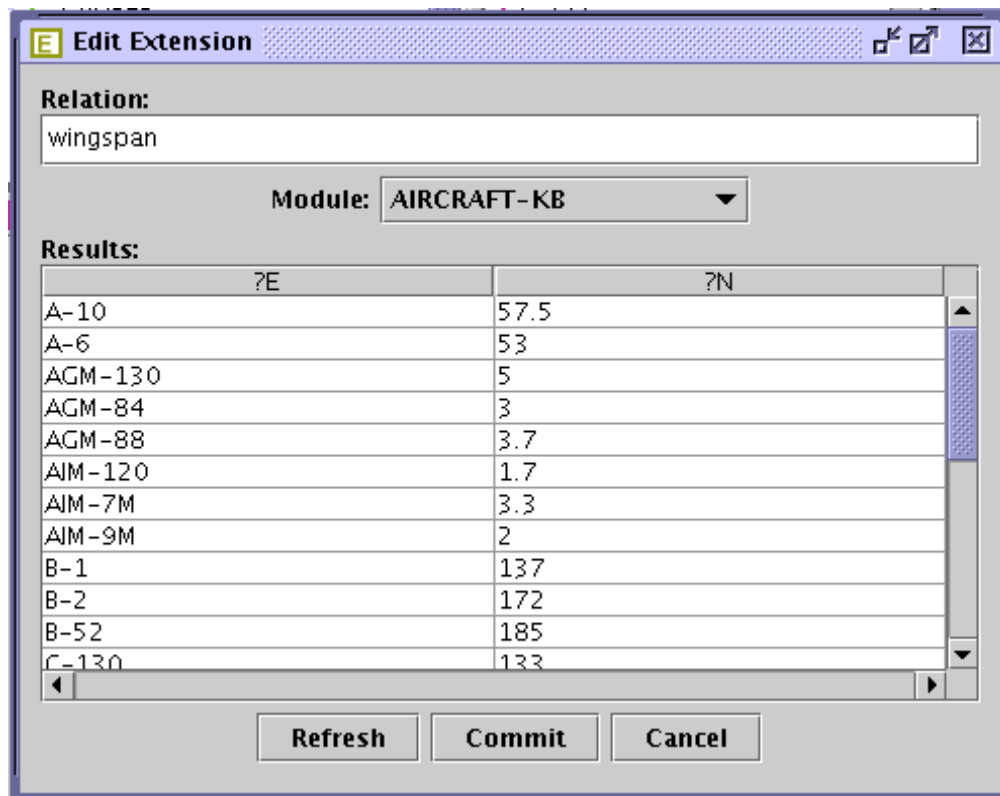


Figure 15: Extension Editor

4.4.7 Extension Editor

The extension editor, shown in Figure 15, allows editing of a concept or relation's extension, and can be opened by right-clicking on a concept or relation in the browser or by selecting the `Query -> Edit Extension` menu item. The extension editor presents a relation's extension as a list of tuples in table format. The user may add new tuples by typing names of instances at the bottom of the table, and may alter existing tuples but double-clicking on a table cell and typing in a new value. Instance name completion is available while typing instance names by typing `CTRL - [right arrow]`. A user may choose to abort the edited extension by clicking the `Cancel` button. If the user clicks the `Commit` button, the relation's extension will be updated by asserting and retracting appropriate propositions.

Query

Query:
(wingspan ?X ?Y)

Module: AIRCRAFT-KB

Results:

?X	?Y
A-10	57.5
A-6	53
AGM-130	5
AGM-84	3
AGM-88	3.7

Saved Queries

Save Current Query

Query Options

☒ Retrieve all results
☐ Retrieve results

Timeout:
Moveout:

Inference Level: NORMAL

Match Mode: STRICT

Minimum Score:
☐ Maximize Score

Maximum Unknowns:
☐ Don't Optimize

Hide Advanced Options

Execute
Continue
Cancel

Figure 16: Query Dialog

4.4.8 Query/Ask

The Query dialog, shown in Figure 16, can be opened by selecting the `Query` -> `Query` menu item, typing `CTRL-Q` or by pressing the query toolbar button. The Query dialog consists of a text area for typing the query, a results table for displaying the results of the query, a query list for selecting pre-saved queries, and an options subpane for configuring various query parameters.

The query input pane supports features similar to that of the proposition editor, including Emacs key bindings, parenthesis matching, and completion. Queries can be executed by hitting `CTRL-[RETURN]` or by clicking on the Execute button at the bottom of the dialog. After a query has executed, results will be displayed in the results table or a “No results found” indicator will flash in the results area. The column headers for the results will display the corresponding free variables in the query. Results may be sorted by clicking on a column header. Doing so will sort the results by using the clicked column as an index. Users may toggle ascending/descending sort order by clicking the header multiple times.

If the query contains no free variables, it is effectively an ASK operation (as opposed to a RETRIEVE operation). In this case, the result will be a truth value, and the column header will be labeled `TRUTH-VALUE`. If the query is the result of a partial retrieve operation, an additional column containing the match score will be displayed.

If the user clicks on a cell in the results table, the topmost browser will be updated to display the selected item. For cases where a partial query was performed, the user may right-click on a query result and select the “Show Explanation” menu item. Selecting this will present an HTML explanation in a separate window. The displayed explanation may contain hyperlinked objects. Clicking on a hyperlinked object will update the topmost browser to display the object.

Users may save frequently-executed queries in a query list by clicking the Save button at the top of the options panel. After clicking save, they will be prompted for a query name. Saved queries will be stored in the preferences file and are represented as XML. Saved queries are stored in the combobox to the left of the save button. Selection of a saved query will prefill the Query dialog with the query and all saved parameters.

All PowerLoom query options are available in the options dialog. These options currently include the query's timeout, moveout, maximum number of unknowns, minimum score, and match mode.

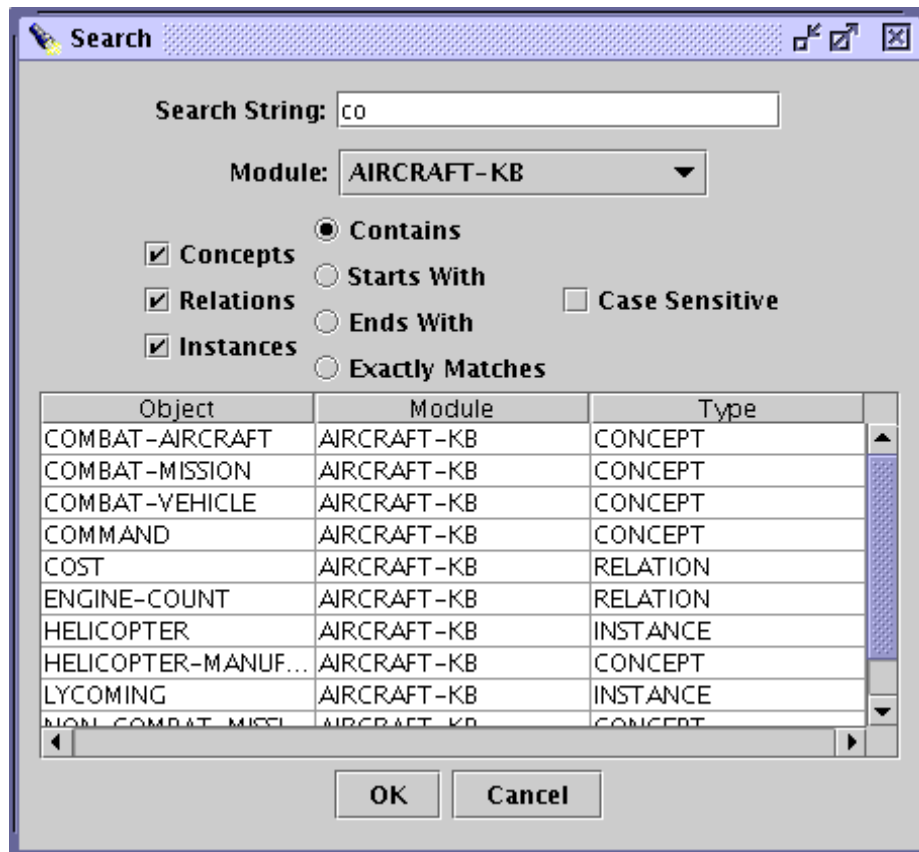


Figure 17: Search Dialog

4.4.9 Search

Users may search for objects in the KB by entering strings which match the name of the object. A search dialog as shown in Figure 17 can be opened by selecting the Query -> Search menu item, typing CTRL-F, or by pushing a search toolbar button inside the browser. If the user pushes a search toolbar button inside a navigation pane, the search dialog will be configured to search for objects associated with the type of object

displayed in the pane. For example pushing the search button inside the concept navigation pane will configure the search dialog to look for concept objects.

Searches may be constrained in several ways. First, the type of module may be specified or the user may specify that the search should be across all modules. Second, the types of objects to be searched is configurable. For example, users may search for concepts and instances, instances only, etc. Finally, users may specify that the objects name must match the beginning or end of the search string, or exactly match the search string.

When the user executes the search by hitting return or selecting the OK button, a list of results is presented. These results are presented in table format, where one column is the name of the retrieved object, another column contains the module that the object resides in, and the final column specifies the type of the object (i.e., concept, instance, etc). As is the case with query results, clicking on a search result item will update the topmost browser to display the selected object.

4.4.10 Console

The console window, as shown in, can be opened by selecting the KB -> Open PowerLoom Console menu item or typing CTRL-P. This opens an internal window, which allows PowerLoom commands to be typed directly and sent to the PowerLoom server. The response generated by PowerLoom is sent back to the GUI and printed below the prompt. This functionality is similar to that of a LISP listener.

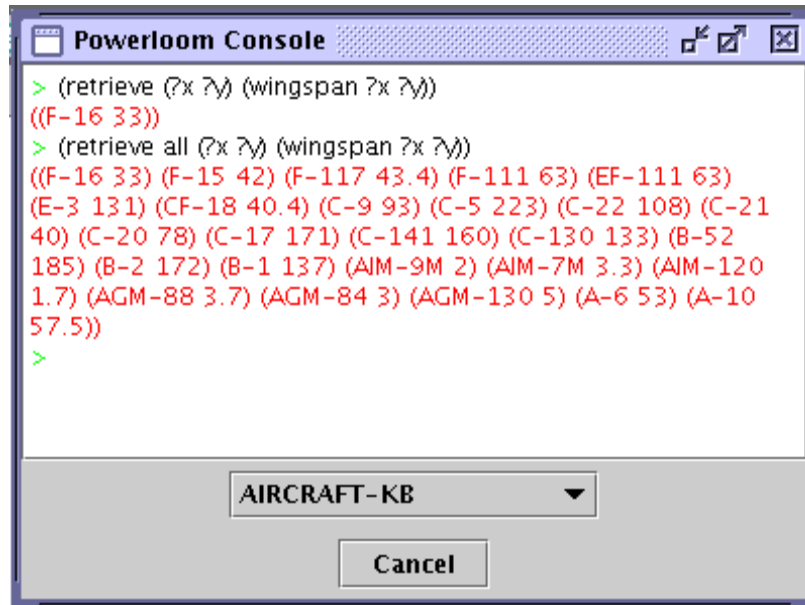


Figure 18: PowerLoom Console

4.4.11 Cut/Copy/Paste/Delete

The PowerLoom GUI supports Cut, Copy, Paste, and Delete operations. These operations can be used to edit text, and in some cases they can be used to edit objects in list or trees. For example, the concept hierarchy can be edited within the browser by selecting a concept, executing a cut operation, selecting another concept, and then executing paste. This sequence of operations will delete the concept from its original position in the hierarchy, and make it a subconcept of the concept that was selected when the paste operation was performed.

We have implemented a robust data transfer framework that is capable of recognizing the types of objects that are being transferred, and the types of potential transfer sources and destinations. This allows the application to prohibit nonsensical data transfers such as cutting a concept in a concept navigation pane and then trying to paste it into a module pane. It also allows data transfer operations to be context sensitive. For example, cutting a concept in a concept navigation pane means that a move operation is being initiated, while cutting a concept in a concept editor's superconcept list means that the concept should be removed from the list. Additionally, copying an object such as a concept, then executing a paste inside a text window will paste the name of the object.

As one would expect, text may be cut/copied/pasted between the GUI and outside applications.

4.5 Future Work

There are many areas that are good candidates for future development efforts, including:

4.5.1 Large KBs

Currently, when a module is selected, the GUI attempts to retrieve all concepts, relations, and instances that are contained in the module. For large knowledge bases, this is clearly infeasible. Some knowledge bases, such as Cyc, contain millions of instances and assertions. We need to develop more sophisticated caching strategies to flush old or rarely-used knowledge from the GUI. Also, we need to develop methods for retrieving fixed-sized chunks of a KB at a time. For example, rather than presenting a list of all instances in a module, we might initially present a fixed number N instances, and display a button labeled “More instances...” which will retrieve N more instances. A similar strategy can be employed for tree representations of hierarchies. At first, the topmost objects in the hierarchy can be retrieved, and as the user expands the tree, knowledge can be retrieved on demand.

4.5.2 Drag/Drop

Adding a drag and drop capability would make ontology editing easier than is currently possible. For example, one concept could be dragged on top of another to move the object from its current position. We believe that the existing data transfer framework could be leveraged to implement a robust drag and drop facility.

4.5.3 Scrapbook

In creating and editing ontologies, it is sometimes desirable to maintain heterogeneous scraps of information. We envision a scrapbook feature where text and objects of various types could be dragged and arranged visually.

4.5.4 Instance cloning

It is often useful to create new instances that are similar to existing instances. We would like to implement a cloning facility in which a wizard-like series of dialogs would step the user through the process of copying information from one object to a new object. For example, the dialogs would prompt the user for propositions to transfer from the old instance to the new instance, and allow the user to modify the propositions in the process of transferring them.

4.5.5 Security

There is virtually no security implementation in the PowerLoom GUI. The GUI client assumes that it is communicating with a trusted host over a secure network. Similarly, the PowerLoom server assumes that it is communicating with a friendly client that has full access to the server. In the future, we need to add security mechanisms which allow clients to be authenticated, and resources on the server to be made accessible to authorized users only. In addition, we need to implement encryption mechanisms so that that clear text is not sent over insecure networks, potentially compromising sensitive data.

4.5.6 Multiple users

Although the client/server model allows multiple GUI clients to concurrently share the same server, there is very weak support for synchronizing clients and ensuring that users don't accidentally step on each other. We need to improve our infrastructure to handle notification of KB updates, add support for transactions and KB locking, and improve our caching mechanisms to detect when the GUI state is out of sync with respect to the server.

5 References

J.F. Allen. Toward a general theory of action and time 1984. *Artificial Intelligence*, 23(2):123--154, 1984.

J. Blythe, Y. Gil, H. Chalupsky, and R.M. MacGregor 2000. Supporting translation among planning agents. Internal Project Report, USC Information Sciences Institute, 2000.

H. Chalupsky and R.M. MacGregor 1999. STELLA – a Lisp-like language for symbolic programming with delivery in Common-Lisp, C++ and Java. In *Proceedings of the 1999 Lisp User Group Meeting*, Berkeley, CA, Franz. Inc. (see also <http://www.isi.edu/isd/LOOM/Stella/index.html>).

V.K. Chaudhri, A. Farquhar, R. Fikes, P.D. Karp, and J.P. Rice 1998. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 600--607, Menlo Park, July 26--30 1998. AAAI Press.

P.R. Cohen, R. Schrag, E. Jones, A. Pease, A. Lin, B. Starr, D. Easter, D. Gunning, and M. Burke 1998. The DARPA High Performance Knowledge Bases project. *Artificial Intelligence Magazine*, 19(4):25--49, 1998.

M.T. Cox and M.M. Veloso 1997. Controlling for unexpected goals when planning in a mixed-initiative setting. In E. Costa and A. Cardoso, editors, *Proceedings of the Eighth Portuguese Conference on Artificial Intelligence (EPIA-97)*, volume 1323 of *LNAI*, pages 309--318, Berlin, 1997. Springer.

Emde, W. 1996. Relational instance-based learning In *Proceedings of the 13th International Conference on Machine Learning*.

R. Fikes, A. Farquhar, and J. Rice 1997. Tools for assembling modular ontologies in Ontolingua. In *Proceedings of the 14th National Conference on Artificial Intelligence*

and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97), pages 436--441, Menlo Park, July 27--31 1997. AAAI Press.

G. Frank, A. Farquhar, and R. Fikes 1999. Building a large knowledge base from a structured source. *IEEE Intelligent Systems*, 14(1):47--54, 1999.

Gebhardt, F. 1997. Survey on structure-based case retrieval. *The Knowledge Engineering Review*, 12(1), 41--58.

M.R. Genesereth 1991. Knowledge interchange format. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 599--600, San Mateo, CA, USA, April 1991. Morgan Kaufmann Publishers.

Gentner, D. and Forbus, K. 1991. MAC/FAC: A model of similarity-based retrieval. In *Proceedings of the Cognitive Science Society*.

Y. Gil 1994. Knowledge refinement in a reflective architecture. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 520--526, Menlo Park, CA, USA, July 31--August 4 1994. AAAI Press.

T.R. Gruber 1993. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199--220, 1993.

E.H. Hovy 1998. Combining and standardizing large-scale, practical ontologies for machine translation and other uses. In *Proceedings of the First International Conference on Language Resources and Evaluation (LREC)*, Granada, Spain, 1998.

Jones, E. 1999. HPKB course of action challenge problem specification, Alphatech Inc.

K. Knight and S.K. Luk 1994. Building a large-scale knowledge base for machine translation. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 773--778, Menlo Park, CA, 1994. AAAI Press.

Kolodner, J. L. 1993. *Case-Based Reasoning*. Morgan Kaufmann.

D. Lenat 1995. CYC: A Large Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11):32--38, November 1995.

- R.M. MacGregor 1991. Inside the LOOM description classifier. *ACM SIGART Bulletin*, 2(3):70--76, 1991.
- D.L. McGuinness, R.E. Fikes, J. Rice, and S. Wilder 2000. An environment for merging and testing large ontologies. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*, San Francisco, CA, 2000. Morgan Kaufmann.
- A. Mulvehill and S. Christey 1995. *ForMAT -- a Force Management and Analysis Tool*. MITRE Corporation, Bedford, MA, 1995.
- Pazzani, M., Mani, S., and Shankle, W. R. 1997. Beyond concise and colorful: Learning intelligible rules. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, 235--238.
- S. C. Shapiro and W. J. Rapaport 1992. The SNePS family. *Computers & Mathematics with Applications*, 23(2--5):243--275, January--March 1992.
- D.C. Smith 1990. *Plisp User's Manual*. Apple Computer, August 1990.
- J. F. Sowa 1992. Conceptual graphs as a universal knowledge representation. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 75--93. Pergamon Press, Oxford, 1992.
- W. Swartout, R. Patil, K. Knight and T. Russ. 1996. Towards distributed use of large-scale ontologies. In *Proceedings of the Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*. 1996
- L. Tesler, H. Enea, and D.C. Smith 1973. The Lisp70 pattern matching system. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 671--676, Stanford, CA, August 1973. William Kaufmann.
- A. Valente, T.A. Russ, R.M. MacGregor, and W.R. Swartout 1999. Building and (re)using an ontology of air campaign planning. *IEEE Intelligent Systems*, 14(1):27--36, 1999.

M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe 1995. Integrating planning and learning. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

M.M. Veloso 1994. *Planning and learning by analogical reasoning*, volume 886 of *LNAI*. Springer, New York, NY, 1994.

Wettschereck, D., Aha, D. W., and Mohri, T. 1997. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11, 273--314.